
QuickTime Video Effects and Transitions Guide

[QuickTime > Video Effects & Transitions](#)



2007-05-03



Apple Inc.
© 2005, 2007 Apple Inc.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, without prior written permission of Apple Inc., with the following exceptions: Any person is hereby authorized to store documentation on a single computer for personal use only and to print copies of documentation for personal use provided that the documentation contains Apple's copyright notice.

The Apple logo is a trademark of Apple Inc.

Use of the "keyboard" Apple logo (Option-Shift-K) for commercial purposes without the prior written consent of Apple may constitute trademark infringement and unfair competition in violation of federal and state laws.

No licenses, express or implied, are granted with respect to any of the technology described in this document. Apple retains all intellectual property rights associated with the technology described in this document. This document is intended to assist application developers to develop applications only for Apple-labeled computers.

Every effort has been made to ensure that the information in this document is accurate. Apple is not responsible for typographical errors.

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Apple, the Apple logo, Chicago, ColorSync, Geneva, Mac, and QuickTime are trademarks of Apple Inc., registered in the United States and other countries.

Simultaneously published in the United States and Canada.

Even though Apple has reviewed this document, APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS DOCUMENT, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS DOCUMENT IS PROVIDED "AS IS," AND YOU, THE READER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY

DEFECT OR INACCURACY IN THIS DOCUMENT, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

Contents

Introduction Introduction to QuickTime Video Effects and Transitions Guide 9

Organization of This Document 9

See Also 10

Chapter 1 How To Add QuickTime Video Effects 11

Effect Tracks 11

Adding Video Effects to a QuickTime Movie 11

 Preparing an Effect for Direct Execution 12

 Executing the Decompression Sequence 12

Creating an Effects Track 13

Creating an Effect Description 13

 Structure of an Effect Description 14

 Adding the Sample to the Media 14

 Required Atoms of an Effects Description 15

 Example: Cross Fade 15

 Creating an Input Map 16

 Parameter Atoms of an Effects Description 16

Working with Source Tracks 17

 Zero-Source Effects 18

 One-Source Effects (Filters) 18

 Two-Source Effects (Transitions) 18

Sources Other Than Video Tracks 20

Using Video Effects Outside a QuickTime Movie 21

Chapter 2 Constructing a Video Effects User Interface 23

Displaying the Effects User Interface Using the High-Level API 23

 Getting a List of Effects 23

 Displaying the Standard Parameters Dialog Box 24

Processing Standard Parameter Dialog Box Events 25

Adding Video Effects Controls to an Existing Dialog Box 27

Creating Your Application's Dialog Box 27

Incorporating Controls From the Standard Parameters Dialog Box 28

Adding a Preview to Your Dialog Box 29

Chapter 3 Built-in QuickTime Video Effects 31

What Each Effect Does 32

Push 32

The SMPTE Video Effects 33

SMPTE Wipe Effects	33
SMPTE Iris Effects	35
SMPTE Radial Effects	37
SMPTE Matrix Effects	40
Video Effects from Apple	42
Alpha Compositor	42
Blend Mode Enum	43
Alpha Gain filter	43
Blur Filter	44
Chroma Key	45
Cloud	45
Color Style	46
ColorSync filter	46
Color Tint filter	47
Edge Detection Filter	48
Emboss Filter	48
Explode	49
Film Noise Filter	49
The Film Fade Enum	51
Fire	51
General Convolution Filter	52
Gradient Wipe	53
HSL Balance Filter	54
Implode	55
Lens Flare	55
RGB Balance Filter	57
Ripple	57
Sharpen Filter	58

Chapter 4 **Creating New Video Effects** 59

What Effects Components Do	59
The Effect Component Interface	60
Supplying Parameter Description Information	61
Implementing the EffectBegin and EffectRenderFrame Functions	62
The EffectBegin function	62
Checking Source and Destination References	62
Reading Parameter Values	64
Tweening Parameter Values	66
The EffectRenderFrame Function	67
Handling Multiple Formats	67
Implementing a Bit-depth Specific Version of Your Algorithm	68
Including the Bit-depth Implementations in Your Effect Code	69
Calling the Effect Implementations from EffectRenderFrame	70
The Sample Effect Component	71
The Dimmer Effect	71

The Standard Effect Framework	71
Structure of the Framework	71
Naming Conventions	72
Writing an Effect Component Using the Framework	72
Synchronous vs. Asynchronous Processing	72
Defining the Number of Sources	72
Adding to the Global Data Structures	73
Preflighting the Blitter	73
Setting the Destination	74
The BlitterRenderFrame function	74
The EffectsFrameClose function	74
Reading the Effect Parameters	74
Implementing your Effect	74
Adding an 'atms' Resource to your Component	75
The Standard Information in an 'atms' Resource	76
The Parameter Information in an 'atms' Resource	76
The Parameter Description Format	78
Parameter Atom Type and ID	78
Special Description Types	79
Groups	79
Enumeration Lists	80
Source Count	81
Parameter Data Type	81
Parameter Alternate Data Type	82
Parameter Data Range	83
Parameter Data Behavior	85
Parameter Data Usage	87
Parameter Data Default Item	88
Tweening Parameters	88
Slide	89
Parameter Descriptions	90
Component-Defined Functions	91
MyEffectSetup	91
MyEffectBegin	92
MyEffectRenderFrame	92
MyEffectCancel	93
MyEffectGetCodeInfo	93
MyEffectGetParameterListHandle	94
MyEffectGetSpeed	94
MyEffectValidateParameters	95

Chapter 5**Video Effects API 97**

Introduction	97
Constants	97
Effects List Atom Names	97

- Effect Action Selectors 98
- Get Options for QTGetEffectsList 99
- Standard Parameter Dialog Box Options 99
- ImageCodecValidateParameters Options 99
- Effect Speed Flag 100
- Data Types 100
 - Parameter Dialog Box Preview Image Specifier 100
 - Effect Source Descriptors 101
 - Effect Frame Description 102
 - The Decompression Parameters Structure 102
- Functions 104

Document Revision History 105

Listings

Chapter 1 **How To Add QuickTime Video Effects 11**

- Listing 1-1 Defining the RunEffect function, which executes one frame of an effect 12
- Listing 1-2 Executing an effect directly by calling the RunEffect function 13
- Listing 1-3 Calling AddMediaSample to add an effect description 14
- Listing 1-4 Adding a kParameterWhatName atom with the value kCrossFadeTransitionType to the QTAtomContainer effectDesc 15

Chapter 2 **Constructing a Video Effects User Interface 23**

- Listing 2-1 An example event loop showing use of QTIsStandardParameterDialogEvent 26
- Listing 2-2 Creating a dialog box and adding effect controls 28

Chapter 4 **Creating New Video Effects 59**

- Listing 4-1 Implementing the GetParameterListHandle function using GetComponentResource 61
- Listing 4-2 Storing information about a new destination frame 63
- Listing 4-3 Checking for source changes 63
- Listing 4-4 Storing information about a new source frame 64
- Listing 4-5 Reading a parameter value 65
- Listing 4-6 Reading a tweened parameter value 65
- Listing 4-7 Tweening parameter values 66
- Listing 4-8 A sample effect algorithm for 16-bit frames 68
- Listing 4-9 Including the 16-bit implementation into the main effect source code 70
- Listing 4-10 Calling pixel format specific versions of the 16-bit effect implementation 70
- Listing 4-11 An example 'atms' atom declaration 75
- Listing 4-12 An example set of parameter description atoms 77
- Listing 4-13 An example group atom from an 'atms' resource definition. 80
- Listing 4-14 An example enumeration list from an 'atms' resource definition 81
- Listing 4-15 Opening the image decompressor component 90

Introduction to QuickTime Video Effects and Transitions Guide

This book introduces you to QuickTime video effects and transitions. You can use effects and transitions to control the visual transition between two sources. Sources can be tracks in a QuickTime movie or they can be graphics worlds. You can use filter effects to visually alter a single source, such as applying a blur or ripple. You can also use free-standing effects, such as a cloud or fire effect, that do not require a source (though they can be composited with other video).

Note: This document was previously titled “Filters, Effects, and Transitions.”

Because visual effects are calculated and executed at runtime, they typically result in a much smaller file than a pre-rendered version of the same effect.

Effects tracks can be created, edited, and used in essentially the same manner as other video tracks. You can “stack” effects by using one effects track as the source for another effect. You can also use an effect as the source for a sprite track, making the fire effect into a sprite, for example.

QuickTime includes over 145 effects, and its extensible architecture allows you to create additional effects of your own if you need them in your application development.

You need to read this document if you are writing an application that creates QuickTime movies and you want to add video effects to those movies, or if you want to use video effects on graphics worlds without creating a QuickTime movie, or if you want to create new video effects of your own.

This document discusses the high-level functions available to you that provide your application with pre-packaged access to the video effects architecture, and are designed to be easy to use and give you access to the most common uses of the QuickTime Video Effects architecture.

The low-level functions provide more complex and comprehensive interfaces to the effects dialog functionality. Using the low-level functions, you can gain more control over the standard parameters dialog box, such as the ability to incorporate user interface elements from the dialog box into your own application-defined dialog box.

Organization of This Document

This document is divided into five chapters:

- [How To Add QuickTime Video Effects](#) (page 11) discusses how QuickTime video effects are implemented and how you can add effects to QuickTime movies.
- [Constructing a Video Effects User Interface](#) (page 23) discusses how to construct a user interface that enables users to select an effect, change its parameters, and preview the results.
- [Built-in QuickTime Video Effects](#) (page 31) discusses SMTPE effects, which are implementations of over 100 standard effects defined by the Society of Motion Picture and Television Engineers, plus the set of effects implemented by Apple Computer, which you can use for a variety of purposes.

- [Creating New Video Effects](#) (page 59) describes how you can create your own video effects. The chapter walks you through the implementation of a sample effect component. The sample effect is built on a framework of code that you can reuse when you implement your own effect component.
- [Video Effects API](#) (page 97) describes the constants, data types, and functions defined in QuickTime that support video effects.

See Also

The following Apple books cover related aspects of QuickTime programming:

- *QuickTime Overview* gives you the starting information you need to do QuickTime programming.
- *QuickTime Movie Basics* introduces you to some of the basic concepts you need to understand when working with QuickTime movies.
- *QuickTime Guide for Windows* provides information specific to programming for QuickTime on the Windows platform.

How To Add QuickTime Video Effects

This chapter discusses how QuickTime video effects are implemented and how you can add effects to QuickTime movies.

The first step in adding effects to a QuickTime movie is to create an effects track. This is accomplished by using the standard QuickTime API for track creation, as explained in this chapter.

You can also use QuickTime video effects to transition between two graphics worlds. Your application does not have to generate a QuickTime movie to use the video effects.

Effect Tracks

QuickTime video effects are implemented as **components**, which are the standard mechanism used to extend QuickTime. Effect components are actually a specialized type of image decompressor component.

To use an effect component in a QuickTime movie, you add an **effect track** to the movie. The size and duration of the track determines the area of the movie that is affected and how long the effect runs.

The effect track has two important attributes: the effect description and the input map. The **effect description** is a sample, added to the media of the effects track, that selects which effect to use and contains the parameters for that effect. The **input map** describes the *sources* that the effect works on. Effect components use whole tracks as sources. A track reference redirects the output of the source track to the effect track. You may need to make new tracks, referencing parts of existing video tracks, to act as sources for your effects.

Once you have arranged your source tracks and added the effects track to your movie, QuickTime automatically executes the effect when the movie plays. QuickTime generates as many frames per second as possible for the effect, so the effect will run as smoothly as the CPU and display hardware of the target machine permit.

You can also use the QuickTime video effects outside the context of a QuickTime movie. You still supply an effect description, but instead of creating an effect track, you write code that executes the individual steps of the effect. For details, see the section [Using Video Effects Outside a QuickTime Movie](#) (page 21).

Adding Video Effects to a QuickTime Movie

This section explains the steps you need to take in order to add video effects to a QuickTime movie. In brief, you proceed as follows:

1. You create and arrange any **source tracks** that will be used by the effect.
2. You add a new **effect track** to your movie: the offset and duration of the track determine when the effect takes place.

3. You create an **effect description** that selects the particular effect you want and supplies values for any parameters the effect has.
4. You create an **input map** that defines which tracks in the movie serve as sources for the effect.
5. Finally, you add the effect description as a new sample to the media of the effect track. As part of this process, you create a **sample description**, which describes the sample being added.

There are high-level routines that can be used to greatly simplify this process. For example, `QTCreateStandardParameterDialog` can automatically obtain a list of available effects, allow the user to choose an effect and set its parameters, and return an effect description for the chosen effect.

Preparing an Effect for Direct Execution

The code that prepares the data structures required to directly execute an effect is broadly similar to the code to set up effects within a QuickTime movie.

You first provide an effect description and sample description for the effect component you are going to use. Then you prepare a decompression sequence that will actually playback the effect.

Executing the Decompression Sequence

With the effect and sample descriptions built and the decompression sequence prepared, you can now execute the effect. The function shown in Listing 1-1 executes a single frame of a decompression sequence.

The parameter `theTime` contains the number of the frame to be executed. The parameter `theNumberOfSteps` contains the total number of frames that will be used to run the effect.

Listing 1-1 Defining the `RunEffect` function, which executes one frame of an effect

```
// Decompress a single step of the effect sequence at time.
OSErr RunEffect(TimeValue theTime, int theNumberOfSteps)
{
    OSErr          err = noErr;
    ICMFrameTimeRecord  frameTime;
    // Set the timebase time to the step of the sequence to be rendered
    SetTimeBaseValue(gTimeBase, theTime, theNumberOfSteps);
    frameTime.value.lo    = theTime;
    frameTime.value.hi    = 0;
    frameTime.scale      = theNumberOfSteps;
    frameTime.base       = 0;
    frameTime.duration   = theNumberOfSteps;
    frameTime.rate       = 0;
    frameTime.recordSize = sizeof(frameTime);
    frameTime.frameNumber = 1;
    frameTime.flags      = icmFrameTimeHasVirtualStartTimeAndDuration;
    frameTime.virtualStartTime.lo = 0;
    frameTime.virtualStartTime.hi = 0;
    frameTime.virtualDuration    = theNumberOfSteps;
    HLock(myEffectDesc);
    DecompressSequenceFrameWhen(gEffectSequenceID,
                                StripAddress(*myEffectDesc),
```

```

        GetHandleSize(myEffectDesc),
        0,
        0,
        nil,
        &frameTime);
    HUnlock(myEffectDesc);
}

```

The code in Listing 1-2 executes this effect in 30 steps.

Listing 1-2 Executing an effect directly by calling the RunEffect function

```

for (currentTime = 0; currentTime < 30; currentTime++)
{
    myErr = RunEffect(currentTime, 30);
    if (myErr != noErr)
        goto bail;
}

```

Creating an Effects Track

The first step in adding effects to a QuickTime movie is to create an effects track. This is accomplished by using the standard QuickTime API for track creation, for example:

```
theEffectsTrack = NewMovieTrack(theMovie, kTrackWidth, kTrackHeight, 0);
```

You then call the `NewTrackMedia` function to add a media to the track. The type of the media for an effects track should always be `VideoMediaType`, and the media should have whatever duration you want the effect to have. Here is a sample call to `NewTrackMedia`:

```
theEffectsMedia = NewTrackMedia(theEffectsTrack, VideoMediaType, 600, nil, 0);
```

Creating an Effect Description

An effect description tells QuickTime which effect to execute and contains the parameters that control how the effect behaves at runtime. You create an effect description by creating an atom container, inserting a QT atom that specifies the effect, and inserting a set of QT atoms that set its parameters.

There are support functions you can call to assist you in this process. `QTCreateStandardParameterDialog` returns a complete effect description that you can use, including user-selected settings; you only need to add `kEffectSourceName` atoms to the description for effects that require sources. At a lower level, `QTGetEffectsList` returns a list of the available effects and `ImageCodecGetParameterList` will return a description of the parameters for an effect, including the default value for each parameter in the form of a QT atom that can be inserted directly into an effect description.

Structure of an Effect Description

An effect description is the sole media sample for an effect track. An effect description is implemented as a `QTAtomContainer` structure, the general QuickTime structure for holding a set of QuickTime atoms. All effect descriptions must contain the set of **required atoms**, which specify attributes such as which effect component to use. In addition, effect descriptions can contain a variable number of **parameter atoms**, which hold the values of the parameters for the effect.

Each atom contains either data or a set of child atoms. If a parameter atom contains data, the data is the value of the parameter, and this value remains constant while the effect executes. If a parameter atom contains a set of child atoms, they typically contain a **tween entry** so the value of the parameter will be interpolated for the duration of the effect.

You assemble an effect description by adding the appropriate set of atoms to a `QTAtomContainer` structure.

You can find out what the appropriate atoms are by making an `ImageCodecGetParameterList` call to the effect component. This fills an atom container with a set of **parameter description atoms**. These atoms contain descriptions of the effect parameters, such as each parameter's atom type, data range, default value, and so on. The default value in each description atom is itself a QuickTime atom that can be inserted directly into your effect description.

You can modify the data in the parameter atoms directly, or let the user set them by calling `QTCreateStandardParameterDialog`, which returns a complete effect description (you need to add `kEffectSourceName` atoms for effects that require sources).

You then add the effect description to the media of the effect track, as described in the section [Adding the Sample to the Media](#) (page 14).

Adding the Sample to the Media

Once you have the sample description prepared, you can call `AddMediaSample` to add the effect description to the media. Listing 1-3 shows a sample call.

Listing 1-3 Calling `AddMediaSample` to add an effect description

```
// Always call BeginMediaEdits before adding sample to a media
BeginMediaEdits(theEffectsMedia);
// Add the sample to the media
AddMediaSample(theEffectsMedia,
               (Handle) theEffectDescription,
               0,
               GetHandleSize((Handle) theEffectDescription),
               600,
               (SampleDescriptionHandle) sampleDescription,
               1,
               0,
               &sampleTime);
// End the media editing session
EndMediaEdits(theEffectsMedia);
```

Required Atoms of an Effects Description

There are several required atoms that an effect description must contain. The first is the `kParameterWhatName` atom, which contains the name of the effect. This specifies which of the available effects to use.

The code snippet shown in Listing 1-4 adds a `kParameterWhatName` atom to the atom container `effectDesc`. The constant `kCrossFadeTransitionType` contains the name of the cross-fade effect. The cross-fade effect is described in detail in [Example: Cross Fade](#) (page 15).

Listing 1-4 Adding a `kParameterWhatName` atom with the value `kCrossFadeTransitionType` to the QTAtomContainer `effectDesc`

```
effectCode = kCrossFadeTransitionType;
QTInsertChild(effectDescription,
              kParentAtomIsContainer,
              kParameterWhatName,
              kParameterWhatID,
              0,
              sizeof(effectCode),
              &effectCode,
              nil);
```

In addition to the `kParameterWhatName` atom, the effect description for an effect that uses sources must contain one or more `kEffectSourceName` atoms. Each of these atoms contains the name of one of the effect's sources. An **input map** is used to map these names to the actual tracks of the movie that are the sources. [Creating an Input Map](#) (page 16) describes how to create the input map.

Example: Cross Fade

```
kCrossFadeTransitionType ('dslv')
```

A “cross fade” or “dissolve” provides a smooth alpha blending between two video sources, changing over time to give a smooth fade out from the first source into the second.

This effect takes a maximum of two sources and has a single parameter.

Use the description below to help you understand what the parameter does. To learn how to use parameter atoms, see [Adding Video Effects to a QuickTime Movie](#) (page 11).

Name	Code	QTAtom Type	Description
Percentage	'pcnt'	<code>kParameterType-DataFixed</code> ; Always a tween	This parameter contains a tween. As the effect progresses, QuickTime renders the frame of the effect indicated by the tween's current value as a percentage of the whole effect. For example, if the tween goes from 0 to 100, the effect renders completely; if the tween goes from 25 to 75, rendering begins 25% into the effect and terminates 75% through the effect.

Creating an Input Map

The input map is another `QTAtomContainer` structure that you attach to the effects track. It describes the sources used in the effect and gives a name to each source. This name is used to refer to the source in the effect description.

An input map works in concert with track reference atoms in the source tracks. A track reference atom of type `kTrackModifierReference` is added to each source track, which causes that source track's output to be redirected to the effects track. An input map is added to the effects track to identify the source tracks and give a name to each source, such as `'srcA'` and `'srcB'`. The effect can then refer to the sources by name, specifying that `'srcB'` should slide in over `'srcA'`, for example.

Parameter Atoms of an Effects Description

In addition to the required atoms, the effects description contains a variable number of parameter atoms. The number and types of parameter atoms vary from effect to effect. For example, the cross fade effect has only one parameter, while the general convolution filter effect has nine. Some effects have no parameters at all, and do not require any parameter atoms. The chapter [Built-in QuickTime Video Effects](#) (page 31) describes the parameters expected by the built-in effects.

You can obtain the list of parameter atoms for a given effect by calling the effect component using the `ImageCodecGetParameterList` function. The parameter description atoms it returns include default settings for each parameter in the form of parameter atoms that you can insert into your effect description.

The `QTInsertChild` function is used to add these parameters to the effect description, as seen in the code example in Listing 1-4 above.

Consider, for instance, the push effect. Its effect description contains a `kParameterWhatName` atom, two `kEffectSourceName` atoms, and two parameter atoms, one of which is a tween.

The `kParameterWhatName` atom specifies that this is a `'push'` effect.

The two `kEffectSourceName` atoms specify the two sources that this effect will use, in this case `'srcA'` and `'srcB'`. The names correspond to entries in the effect track's input map.

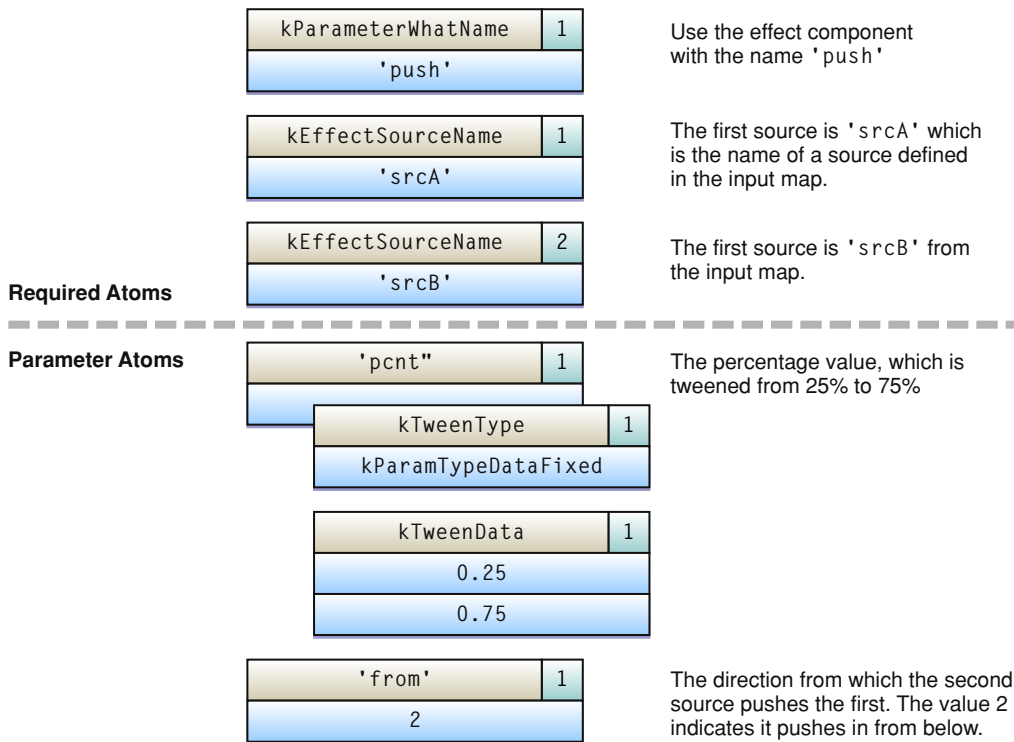
The `'pnt'` parameter atom defines which frames of the effect are shown. This parameter contains a tween entry, so that the value of this parameter is interpolated as the effect runs. The interpolation of the `'pnt'` parameter causes consecutive frames of the effect to be rendered, creating the push effect.

The `'from'` parameter determines the direction of the push. This parameter is set from an enumeration list, with 2 being defined as the bottom of the screen.

In this example, the source `'srcB'` will push in from the bottom, covering the source `'srcA'`.

The `'pnt'` parameter is normally tweened from 0 to 100, so that the effect renders completely, from 0 to 100 percent. In this example, the `'pnt'` parameter is tweened from 25 to 75, so the effect will start 25% of the way through (with `'srcB'` already partly on screen) and finish 75% of the way through (with part of `'srcA'` still visible).

Figure 1-1 shows the set of atoms that must be added to the entry description.



An important property of effect parameters is that most can be tweened (and some must be tweened). Tweening is QuickTime's general purpose interpolation mechanism. For many parameters, it is desirable to allow the value of the parameter to change as the effect executes.

Working with Source Tracks

You will probably need to do some track-level editing on the tracks your effect will use as sources, depending mainly on the type of effect you choose. There are different considerations for an effect that requires no sources, such as the cloud effect, an effect that requires one source, such as the blur filter, or an effect that requires two or more sources, such as a wipe effect.

An effect component can use any track with video output as a source. Effects are normally applied to video tracks, but a sprite track or a text track can also be a source for an effect. It is even possible to "stack" effects, simply by making one effect track a source for another. Stacking effects this way will make serious real-time demands on the target system's processor, however, and the end result may not be satisfactory on all machines.

Generally speaking, an effect uses an entire video track as a source. A track reference atom of type `kTrackModifierReference` is added to the source track, causing the output of the source track to be redirected through the effect.

To make a video track into a source track, for example, you call the `AddTrackReference` function, as shown below.

```
long addedIndex;
AddTrackReference(theEffectTrack, theSourceTrack,
```

```
kTrackModifierReference, &addedIndex);
```

The `kTrackModifierReference` track reference sends all of the source track's output to an effect track, even if the effect track has a smaller duration than the source. If you want to apply an effect to just part of a track, you need to create a new track that references the portions of source media that you want the effect to use. This is explained in more detail in the examples below.

Zero-Source Effects

Effects that don't require a source, such as the fire or cloud effect, are free-standing special effects that can be added anywhere in a movie. Just set the offset and duration of the effects track to the part of the movie where you want the effect to appear. If there is already an active video track at that point in the movie, you control the interaction between the video track and the effects track in the usual ways: putting one track in front of the other, using an alpha channel to allow one track to be partly visible through the other, and so on.

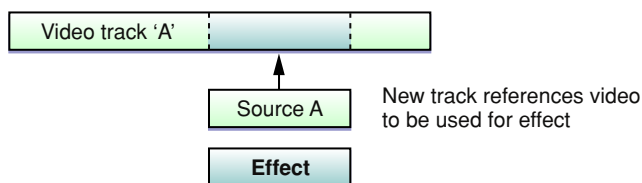


One-Source Effects (Filters)

Effects that require a source, such as a blur filter, steal the output of a video track by using an input map. The video track's output is sent to the effects track, and the effect component acts as a special kind of codec to convert the video into the desired effect.

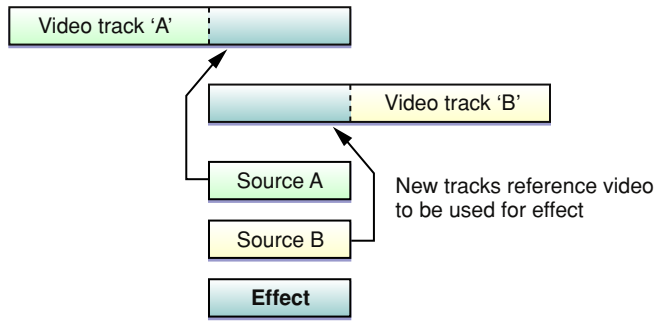
If you want to apply a filter effect to a whole video track, create an effects track with the same offset and duration as the source track. The input map does the rest.

If you want to apply a filter effect to part of a video track, make a new track that references the desired part of the video, then create an effects track with the same offset and duration as this new track. The new track is the source for the effect. You normally want to put the effects track in front of the original track.



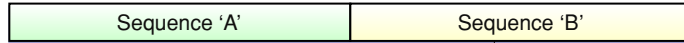
Two-Source Effects (Transitions)

An effect that requires two sources, such as a wipe transition, requires some forethought when setting up the source tracks. If you want to create a transition effect between two video clips, you normally make each clip a separate video track, setting the offset of the second track so that it overlaps the end of the first track by the duration of the transition (see the illustration below). You then make two new tracks that reference the end of the first clip and the beginning of the second clip. These new tracks will act as sources for the effect. The effects track and both source tracks should share the same offset and duration, which correspond to the overlap between the two original tracks, as shown in Figure 1-4.

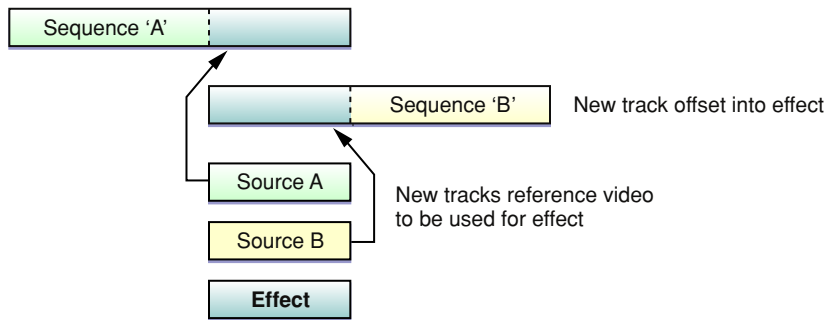


If you want to insert an effect between a sequence of images that now follow each other directly, you face some choices. One choice is to create an effect that overlaps the end of the first sequence with the beginning of the second sequence, making the movie shorter by the length of the effect. This is the usual approach to take, and is illustrated in Figure 1-5.

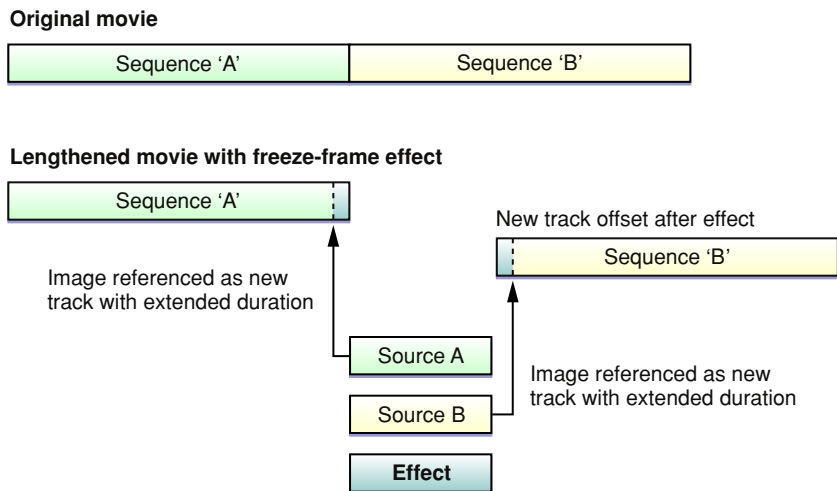
Original movie



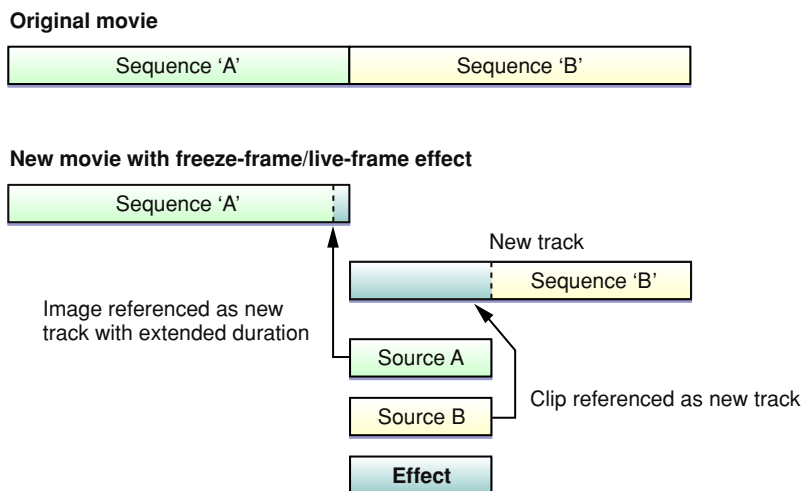
Shortened movie with transition effect



Alternately, you can create an effect that transitions between the last image of the first sequence and the first image of the following sequence, freezing both sequences during the transition, and making the movie longer by the duration of the effect, as shown in Figure 1-6. You would normally use this approach to create a transition between two still images, and you could then restore the movie length by shortening the duration of one or both images.



Either shortening or lengthening a movie can cause problems, particularly if there is a single continuous sound track. To add a transition between two elements that are now sequential, without changing the length of the movie or removing part of the original tracks, you create one frozen source track and one moving source track. In traditional movie editing, this type of transition freezes the first clip, while the second clip is active during the transition, but you can reverse this for a more unusual effect. This technique is illustrated in Figure 1-7.



Sources Other Than Video Tracks

Any track that produces video output, such as a sprite track, a text track, or another effect track, can be used as the source for an effect. Generally speaking, you use these track types as sources in the same way you use a video track, but some special considerations apply.

If you use sprite tracks as sources, and the sprites can move as a result of user interaction, it may be difficult to accurately predict what a transition effect will look like when it executes at run time.

If you “stack” effects, by using an effects track as a source, the target system must have enough speed to render the source frames for the original effect, then render the effect that is acting as a source, then render the stacked effect, while maintaining a reasonable frame rate. For example, the target system might need to decode a pair of Sorenson frames from two video source tracks, apply a cross-fade transition effect between them, and then apply a ripple effect to the final output. This will only give good results on a fast system.

Similarly, you can use an effect as the source for a sprite track, making the cloud effect a sprite, for example. Build the effect description and sample description as described later in this section, then add the effect description to the sprite track as a media sample, just as you would add an ordinary video sample.

Using Video Effects Outside a QuickTime Movie

Adding video effects to a QuickTime movie, as discussed in this chapter, is straightforward enough. You can also use QuickTime video effects to transition between two graphics worlds. Your application does not have to generate a QuickTime movie to use the video effects. This section deals with the task of running an effect that transitions between two graphics worlds. The general principles also apply to filtering a single image held in a graphics world.

Preparing to execute an effect outside the context of a QuickTime movie is similar to preparing to add a video effect to a movie: you provide an effect description and a sample description. The main difference is that instead of building an input map to describe the sources used by the effect, you use the function `CDSequenceNewDataSource` to use a graphics world as the source for the effect.

As well as setting up the effect, you must provide code to run it, since QuickTime cannot directly control the playback of the effect. Because effects are just a type of image decompressor, the code to execute an effect is the same code you would use to decompress and display an image sequence.

Constructing a Video Effects User Interface

This chapter discusses how to construct a user interface that enables users to select an effect, change its parameters, and preview the results.

If your application creates QuickTime movies with video effects, you need to provide the user with a way to choose an effect, adjust its parameters, and preview the results. Although you are free to write your own code for these user interface tasks, you are encouraged to use the APIs that QuickTime provides.

QuickTime provides a standard dialog box (called the **standard parameters dialog box**) that allows the user to select an effect, choose values for its parameters, and preview the effect. Using this dialog box means that your users see an interface that is standard across applications. In addition, if effects parameters change in the future, your application will not need to be rebuilt to use them.

In most applications, it is appropriate to show the standard parameters dialog box to let users choose and customize effects. QuickTime provides a set of high-level API functions that you can call to do this. [Displaying the Effects User Interface Using the High-Level API](#) (page 23) explains the use of these functions.

In some cases, you might need greater control over the effects user interface than these high-level functions provide. You might, for example, want to add the controls from the standard dialog box to one of your application's own dialog boxes. QuickTime provides a set of low-level APIs that let you do this. They give you greater control and flexibility than the high-level functions, but they are more complex to use. These low-level functions are discussed in [Adding Video Effects Controls to an Existing Dialog Box](#) (page 27).

Displaying the Effects User Interface Using the High-Level API

This section describes the set of functions you call to invoke the standard parameters dialog box in your applications.

Getting a List of Effects

Before your application presents the standard parameters dialog box to users, you will probably want to build a list of the effects that are available. QuickTime provides the `QTGetEffectsList` function to do this for you. This returns a `QTAtomContainer` structure that contains a list of all the effects currently available.

Important: The `QTGetEffectsList` function can take several seconds to execute, so you should typically call it only once when your application is launched (or after a pair of suspend and resume events).

You can remove effects from the returned list if you want to restrict the set of effects the user can choose from.

If you present the standard parameters dialog box without providing a list of effects, the dialog function gets the list of available effects automatically. This can save you a few lines of code, but your users must wait while QuickTime searches for available effects and generates the list every time you open the dialog box.

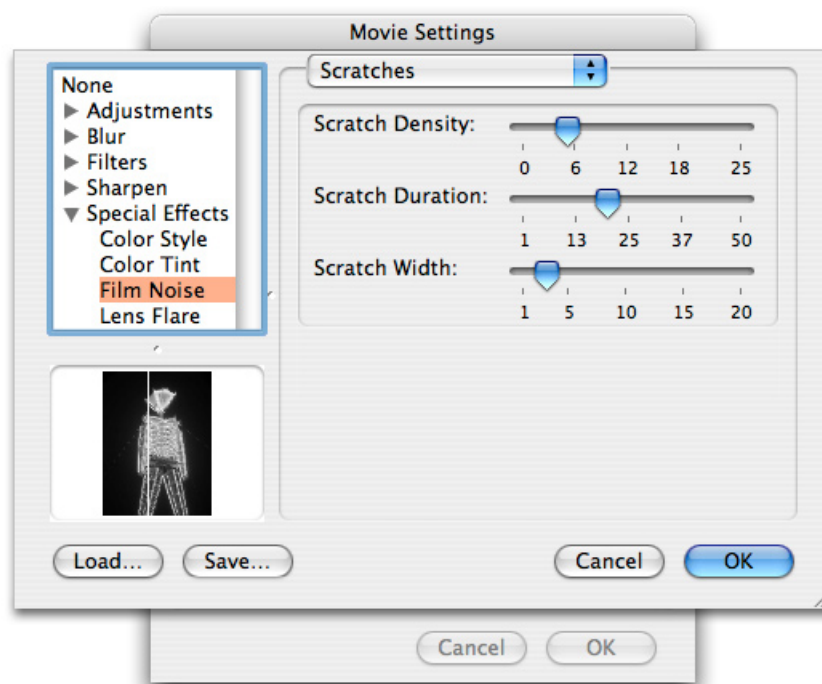
Displaying the Standard Parameters Dialog Box

Use the `QTCreateStandardParameterDialog` function to display the standard parameters dialog box. This allows the user to choose an effect, adjust the settings for that effect, and see a preview of the effect with the selected settings. The use of this function is described below.

When you call the `QTCreateStandardParameterDialog` function, QuickTime creates a standard parameters dialog box. The contents of the dialog box vary, depending on the list of effects you pass to the function and the set of parameters for the currently chosen effect.

Calling `QTCreateStandardParameterDialog` does not immediately display the dialog box; it only prepares it for display. The dialog box is shown the first time you call `QTIsStandardParameterDialogEvent` to process events for the dialog box. Event handling is described in [Processing Standard Parameter Dialog Box Events](#) (page 25).

The standard parameters dialog box for Apple's film noise effect is shown in Figure 2-1.



Notice that the dialog box has three main sections. In the upper-left corner is a scrolling list of all the effect components. To the right of this are the parameters of the chosen effect. As you select different items in the list of effects, the parameters change appropriately. There is also an effect preview area below the list of effects. This shows a preview of the chosen effect and parameter settings.

The user interface for setting the value of a parameter may be a slider, as shown in the example above, a set of radio buttons, an editable text field, or any of several other interfaces specified in the parameter description's `kParameterDataBehavior` field.

For parameters that are always tweens, the user is presented with a starting and ending value. For parameters that can be tweened optionally, the dialog box presents the user with a single value by default. In order to set such a parameter to a tweened value, the user must hold down the Option key when selecting an effect.

Note that your application does not need to know what effects are available, what their parameters are, or what kind of control to use when setting a parameter. All these details are handled by the standard dialog box function.

The function call to create and display this dialog box is

```
QTCreateStandardParameterDialog(theEffectList,
                               theEffectParameters,
                               0,
                               &createdDialogID);
```

The variable `theEffectList` holds the list of effects returned by the `QTGetEffectsList` function. You can also pass `nil` for this value, in which case `QTCreateStandardParameterDialog` calls `QTGetEffectsList` to generate the list of all the currently installed effects, then shows these effects. On input, `theEffectParameters` contains a `QTAtomContainer` structure that holds the initial values of the effect's parameters. In most cases, you should pass an empty `QTAtomContainer` structure as this argument, in which case the default values of each effect are shown.

The third argument specifies how to deal with parameters that can be tweened. Passing in 0 selects the default behavior, which allows the user to set a starting and ending value for parameters that must be tweens, but only allows the user to set a single value for parameters that are optionally tweens.

When the user selects the dialog box's OK button, the chosen effect's parameter values are returned in `theEffectParameters`. The parameter values are returned in a `QTAtomContainer` structure that you can use as an effect description. You will need to add `kEffectSourceName` atoms for effects that use one or more sources.

The `createdDialog` argument returns an ID number that is passed to the other functions that deal with the standard parameters dialog box. This is explained in detail in the next section.

Processing Standard Parameter Dialog Box Events

Once the dialog box has been created, you must process the events sent to it using an event loop. You repeatedly call `WaitNextEvent` and pass the events returned through the `QTIsStandardParameterDialogEvent` function.

The `QTIsStandardParameterDialogEvent` function checks each event to see if it relates to the standard parameters dialog box. You should continue to handle events that are not related to the standard parameters dialog box as usual.



Warning: You should not use the `ModalDialog` function to process events for a standard parameters dialog box. The `ModalDialog` function is not guaranteed to work correctly in all circumstances with a standard parameters dialog box.

You pass the event record returned from `WaitNextEvent` to the `QTIsStandardParameterDialogEvent` function, then check the return value to find out how the events was handled. Common return values are shown in the table below.

Term	Definition
<code>noErr</code>	The event was related to the standard parameters dialog box and was completely processed. Your application should not process this event, instead it should poll <code>WaitNextEvent</code> again.
<code>featureUnsupported</code>	The event was not related to the standard parameters dialog box. Your application should process the event in the normal way.
<code>codecParameter-DialogConfirm</code>	The user clicked the OK button in the standard parameters dialog box. Your application should call <code>QTDismissStandardParameterDialog</code> to close the dialog box. The values chosen by the user are put into the atom container you passed in the <code>parameters</code> parameter when you called <code>QTCreateStandardParameterDialog</code> . This atom container now holds an effect description that is ready to insert into an effects track (it may require one or two <code>kEffectSourceName</code> atoms to be complete).
<code>userCanceledErr</code>	The user clicked the Cancel button in the standard parameters dialog box. Your application should call <code>QTDismissStandardParameterDialog</code> to close the dialog box.

The `QTIsStandardParameterDialogEvent` function may also return error codes, such as memory errors.

Your application should only process the event returned from `WaitNextEvent` if `QTIsStandardParameterDialogEvent` returns an error or a `featureUnsupported` code.

The code in Listing 2-1 is an example event loop showing how `QTIsStandardParameterDialogEvent` is used.

Listing 2-1 An example event loop showing use of `QTIsStandardParameterDialogEvent`

```
while (result == noErr)
{
    EventRecord    theEvent;
    WaitNextEvent(everyEvent, &theEvent, 0, nil);
    result = QTIsStandardParameterDialogEvent(&theEvent,
                                              createdDialogID);

    switch (result)
    {
        case featureUnsupported:
        {
            result = noErr;
            switch (theEvent.what)
            {
                case updateEvt:
```

```

        BeginUpdate((WindowPtr) theEvent.message);
        EndUpdate((WindowPtr) theEvent.message);
        break;
    }
    break;
}

case codecParameterDialogConfirm:
case userCanceledErr:
    QTDismissStandardParameterDialog(createdDialogID);
    createdDialogID = nil;
    break;
}
}
}
}

```

Adding Video Effects Controls to an Existing Dialog Box

In most circumstances, it is best to use high-level functions to create a user interface for video effects. However, if you need finer control over the way the interface is presented, QuickTime provides a set of low-level API functions to assist you.

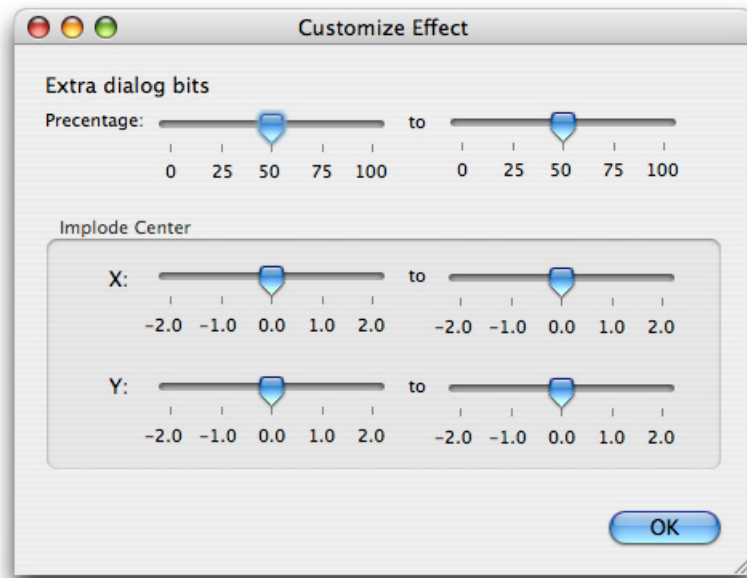
For example, you may want to incorporate the controls from the standard parameters dialog box into an existing dialog box that your application displays, rather than using a completely independent dialog box. This would be necessary if you wanted a single dialog box that allowed users to customize an effect and also specify its duration. To do this, you would create a dialog box with a control for the duration of the effect and then, at runtime, you would add the customization controls from the standard parameters dialog box by calling the `ImageCodecCreateStandardParameterDialog` function.

Note: The low-level APIs follow the same naming convention as their high-level counterparts, except that the low-level versions begin with `ImageCodec`, whereas the high-level versions begin with `QT`.

Creating Your Application's Dialog Box

First create the dialog box that will incorporate the controls from the standard parameters dialog box. This dialog box must contain a user item that is large enough to hold the controls that will be added. For example, if you are using 12-point Chicago as your dialog box font, the user item should be 250 pixels wide and 300 pixels high. If you are using 9-point Geneva, the user item should be 150 pixels wide and 200 pixels high.

Figure 2-2 shows a dialog box in a running application after a custom effects control has been incorporated.



Note: You should make sure that the dialog box resource is marked as not initially visible. You then call a QuickTime function to add the effects controls from the standard parameter dialog box to the application's dialog box before showing it.

Incorporating Controls From the Standard Parameters Dialog Box

Once you have defined the resource for your dialog box, you are ready to add code to your application to create the dialog box and add the effects controls to it. First, call `GetNewDialog` in the usual way to create an instance of the dialog box. Then call `ImageCodecCreateStandardParameterDialog` to incorporate the controls from the standard parameter dialog box into the dialog box. You then show the dialog box in the usual way.

The code in Listing 2-2 shows these steps.

Listing 2-2 Creating a dialog box and adding effect controls

```
movableModalDialog = GetNewDialog(kExtraDialogID, nil, (WindowPtr) -1);
if (movableModalDialog!=nil)
{
    // Add the user interface elements from the standard parameters
    // dialog box to the modal dialog box just created
    ImageCodecCreateStandardParameterDialog(gCompInstance,
                                           parameterDescription,
                                           gEffectSample,
                                           pdOptionsModelDialogBox,
                                           movableModalDialog,
                                           kExtraUserItemID,
                                           &createdDialogID);

    // Now show the dialog box and make it the default port
    ShowWindow(movableModalDialog);
}
```

```

        SelectWindow(movableModalDialog);
        SetPort(movableModalDialog);
    }

```

The call to `ImageCodecCreateStandardParameterDialog` shows how to pass the existing dialog box (`movableModalDialog`) and the item number of the user item (`kExtraUserItemID`) that will be replaced with the controls from the standard parameter dialog box.

Once the effect controls have been added, the dialog box is shown, selected, and made the default port.

The rest of the code needed to handle the dialog box is largely the same as dealing with a stand-alone parameters dialog box. You pass every non-null event returned from `WaitNextEvent` through `ImageCodecIsStandardParameterDialogEvent`, and continue processing events yourself only if it returns `featureUnsupported`.

You track `MouseDown` events sent to the dialog box as usual. When the user clicks the OK button in the dialog box, you need to retrieve the values from the incorporated standard parameters dialog box. To do this, call the function `ImageCodecStandardParameterDialogDoAction` with the `pdActionConfirmDialog` action selector. This retrieves an effect description that describes the parameter values the user has chosen. You then call `ImageCodecDismissStandardParameterDialog` to dispose of the incorporated elements. After this is done, you call `DisposeDialog` to correctly dispose of the application's dialog box.

Adding a Preview to Your Dialog Box

You may have noticed that the incorporated control shown in Figure 2-2 does not include a preview, as the standard parameters dialog box does (Figure 2-1).

In order for your dialog box to show a preview of the effect, include another user item in the application's dialog box to contain the preview movie clip. Then call the `ImageCodecStandardParameterDialogDoAction` function with the `pdActionSetPreviewUserItem` action selector, as shown in the following code snippet:

```

myErr = ImageCodecStandardParameterDialogDoAction(gCompInstance,
                                                gEffectsDialog,
                                                pdActionSetPreviewUserItem,
                                                (void *) kPreviewUserItemID);

```

This makes the user item whose item number is `kPreviewUserItemID` (an application-defined constant in this example) the previewer for your dialog box.

You can use the `ImageCodecStandardParameterDialogDoAction` function to perform a number of similar customizations.

Built-in QuickTime Video Effects

QuickTime includes a set of built-in video effects. There are two classes of effects provided for your use, discussed in this chapter:

- The SMPTE effects, which are implementations of over 100 standard effects defined by the Society of Motion Picture & Television Engineers.
- A set of effects implemented by Apple Computer, which you can use for a variety of purposes.

The following video effects are built into QuickTime, including the standard SMPTE effects and several effects from Apple Computer:

- [The SMPTE Video Effects](#) (page 33)
- [SMPTE Wipe Effects](#) (page 33)
- [SMPTE Iris Effects](#) (page 35)
- [SMPTE Radial Effects](#) (page 37)
- [SMPTE Matrix Effects](#) (page 40)
- [Video Effects from Apple](#) (page 42)
- [Alpha Compositor](#) (page 42)
- [Alpha Gain filter](#) (page 43)
- [Blur Filter](#) (page 44)
- [Chroma Key](#) (page 45)
- [Cloud](#) (page 45)
- [Color Style](#) (page 46)
- [ColorSync filter](#) (page 46)
- [Color Tint filter](#) (page 47)
- [Example: Cross Fade](#) (page 15)
- [Edge Detection Filter](#) (page 48)
- [Emboss Filter](#) (page 48)
- [Explode](#) (page 49)
- [Film Noise Filter](#) (page 49)
- [Fire](#) (page 51)
- [General Convolution Filter](#) (page 52)
- [Gradient Wipe](#) (page 53)
- [HSL Balance Filter](#) (page 54)
- [Implode](#) (page 55)

- [Lens Flare](#) (page 55)
- [Push](#) (page 32)
- [RGB Balance Filter](#) (page 57)
- [Ripple](#) (page 57)
- [Sharpen Filter](#) (page 58)
- [Slide](#) (page 89)

What Each Effect Does

The next sections describe what each effect does. They also define the effect name and the parameter atoms you need in order to create an effect description atom container to implement each effect. For details on how to insert an effect into your movie or application, see [Adding Video Effects to a QuickTime Movie](#) (page 11) and [Using Video Effects Outside a QuickTime Movie](#) (page 21).

Push

```
kPushTransitionType ('push')
```

In a push effect, one source image replaces another with both images moving at the same time. For example, source A would typically occupy the entire frame, then source B would push in from the right while source A slides out to the left, as if source B were pushing source A out of the frame. Unlike the slide effect, both sources are moving. The push effect executes from one of four fixed directions: top, right, bottom, or left.

The push effect takes a maximum of two sources and has two parameters.

Use the descriptions below to help you understand what the parameters do. To learn how to use parameter atoms, see [Adding Video Effects to a QuickTime Movie](#) (page 11).

Name	Code	QTAtom Type	Description
Percentage	'pcnt'	kParameterType-DataFixed; Always a tween	This parameter contains a tween. As the effect progresses, QuickTime renders the frame of the effect indicated by the tween's current value, as a percentage of the whole effect. For example, if the tween goes from 0 to 100, the effect renders completely; if the tween goes from 25 to 75, rendering begins 25% into the effect and terminates 75% through the effect.
From direction	'from'	kParameterType-DataEnum	Contains one of four directions from which source B will replace source A: top, right, bottom, or left.

The 'from' direction parameter can contain the following values:

- Top

- Right
- Bottom
- Left

The SMPTE Video Effects

The SMPTE effects are available in four separate effect components, divided by the type of effect they implement.

There are Wipe effects, Iris effects, Radial effects, and Matrix effects.

SMPTE Wipe Effects

```
kWipeTransitionType('smp')
```

This effect is an implementation of the 34 wipes from ANSI/SMPTE 258M-1993, plus two Apple-defined wipes that choose a random effect. These are a series of masking or “reveal” type wipes that take place between two sources. For full definitions of these 34 wipes and what they look like, refer to the SMPTE documentation.

The SMPTE wipe effects take two sources and seven parameters.

Use the descriptions below to help you understand what the parameters do. To learn how to use parameter atoms, see [Adding Video Effects to a QuickTime Movie](#) (page 11).

Name	Code	QTAAtom Type	Description
Percentage	'pcnt'	kParameterType-DataFixed; Always a tween	This parameter contains a tween. As the effect progresses, QuickTime renders the frame of the effect indicated by the tween's current value, as a percentage of the whole effect. For example, if the tween goes from 0 to 100, the effect renders completely; if the tween goes from 25 to 75, rendering begins 25% into the effect and terminates 75% through the effect.
Wipe ID	'wpID'	kParameterTypeDataEnum	The SMPTE ID for the effect. By setting this parameter, you control which of the 47 available wipes is used. See the enumeration list below.
Soft border	'soft'	kParameterType-DataBitField	If this parameter contains <code>true</code> , the border drawn around the second source is blurred.
Border width	'widt'	kParameterType-DataFixed; Can be a tween	The width, in pixels, of a border that is drawn around the second source.

Name	Code	QTAtom Type	Description
Border color	'bc1r'	kParameterType-DataRGBValue; Can be a tween	The RGB color of the border around the second source.
Horizontal repeat	'hori'	kParameterType-DataLong; Can be a tween	The number of horizontal repeats of the effect that are executed in a single source.
Vertical repeat	'vert'	kParameterType-DataLong; Can be a tween	The number of vertical repeats of the effect that are executed in a single source.

The Wipe ID parameter can take the following values:

Value	Description
1	Slide horizontal
2	Slide vertical
3	Top left
4	Top right
5	Bottom right
6	Bottom left
7	Four corner
8	Four box
21	Barn vertical
22	Barn horizontal
23	Top center
24	Right center
25	Bottom center
26	Left center
41	Diagonal left down
42	Diagonal right down
43	Vertical bow tie
44	Horizontal bow tie
45	Diagonal left out
46	Diagonal right out

Value	Description
47	Diagonal cross
48	Diagonal box
61	Filled V
62	Filled V right
63	Filled V bottom
64	Filled V left
65	Hollow V
66	Hollow V right
67	Hollow V bottom
68	Hollow V left
71	Vertical zig zag
72	Horizontal zig zag
73	Vertical barn zig zag
74	Horizontal barn zig zag
409	Random effect (One of the 133 SMPTE effects is chosen at random)
501	Random wipe (One of the 34 SMPTE wipe effects is chosen at random)

SMPTE Iris Effects

```
kIrisTransitionType('smp2')
```

This effect is an implementation of the 26 iris effects from ANSI/SMPTE 258M-1993, plus two Apple-defined wipes that choose a random effect. These are a series of “reveal” type effects that take place between two sources. For full definitions of these 26 iris effects and what they look like, refer to the SMPTE documentation.

The SMPTE iris effects take two sources and seven parameters.

Use the descriptions below to help you understand what the parameters do. To learn how to use parameter atoms, see [Adding Video Effects to a QuickTime Movie](#) (page 11).

Name	Code	QTAtom Type	Description
Percentage	'pcnt'	kParameterType-DataFixed; Always a tween	This parameter contains a tween. As the effect progresses, QuickTime renders the frame of the effect indicated by the tween's current value, as a percentage of the whole effect. For example, if the tween goes from 0 to 100, the effect renders completely; if the tween goes from 25 to 75, rendering begins 25% into the effect and terminates 75% through the effect.
Wipe ID	'wpID'	kParameterTypeDataEnum	The SMPTE ID for the effect. By setting this parameter, you control which of the 26 available iris effects is used. See the enumeration list below.
Soft border	'soft'	kParameterType-DataBitField; Can be a tween	If this parameter contains <code>true</code> , the border drawn around the second source is blurred.
Border width	'width'	kParameterType-DataFixed; Can be a tween	The width, in pixels, of a border that is drawn around the second source.
Border color	'bclr'	kParameterType-DataRGBValue; Can be a tween	The RGB color of the border around the second source.
Horizontal repeat	'hori'	kParameterTypeDataLong; Can be a tween	The number of horizontal repeats of the effect that are executed in a single source.
Vertical repeat	'vert'	kParameterTypeDataLong; Can be a tween	The number of vertical repeats of the effect that are executed in a single source.

The Wipe ID parameter can take the following values:

Value	Description
101	Rectangle
102	Diamond
103	Triangle
104	Triangle right
105	Triangle upside down
106	Triangle left
107	Arrowhead
108	Arrowhead right
109	Arrowhead upside down

Value	Description
110	Arrowhead left
111	Pentagon
112	Pentagon upside down
113	Hexagon
114	Hexagon side
119	Circle
120	Oval
121	Oval side
122	Cat eye
123	Cat eye side
124	Round rect
125	Round rect side
127	4 point star
128	5 point star
129	6 point star
130	Heart
131	Keyhole
409	Random effect (One of the 133 SMPTE effects is chosen at random)
502	Random iris (One of the 26 SMPTE iris effects is chosen at random)

SMPTE Radial Effects

```
kRadialTransitionType ('smp3')
```

This effect is an implementation of the 39 radial effects from ANSI/SMPTE 258M-1993, plus two Apple-defined wipes that choose a random effect. These are a series of radial “reveal” type effects that take place between two sources. For full definitions of these 39 radial effects and what they look like, refer to the SMPTE documentation.

The SMPTE radial effects take two sources and seven parameters.

Use the descriptions below to help you understand what the parameters do. To learn how to use parameter atoms, see [Adding Video Effects to a QuickTime Movie](#) (page 11).

Name	Code	QTAtom Type	Description
Percentage	'pcnt'	kParameterType-DataFixed; Always a tween	This parameter contains a tween. As the effect progresses, QuickTime renders the frame of the effect indicated by the tween's current value, as a percentage of the whole effect. For example, if the tween goes from 0 to 100, the effect renders completely; if the tween goes from 25 to 75, rendering begins 25% into the effect and terminates 75% through the effect.
Wipe ID	'wpID'	kParameterType-DataEnum	Contains the SMPTE ID for the effect. By setting this parameter, you control which of the 39 available radial effects is used. See the enumeration list below.
Soft border	'soft'	kParameterType-DataBitField	If this parameter contains <code>true</code> , the border drawn around the second source is blurred.
Border width	'widt'	kParameterType-DataFixed; Can be a tween	The width, in pixels, of a border that is drawn around the second source.
Border color	'bclr'	kParameterType-DataRGBValue; Can be a tween	The RGB color of the border around the second source.
Horizontal repeat	'hori'	kParameterType-DataLong; Can be a tween	The number of horizontal repeats of the effect that are executed in a single source.
Vertical repeat	'vert'	kParameterType-DataLong; Can be a tween	The number of vertical repeats of the effect that are executed in a single source.

The Wipe ID parameter can take the following values:

Value	Description
201	Rotating top
202	Rotating right
203	Rotating bottom
204	Rotating left
205	Rotating top bottom
206	Rotating left right
207	Rotating quadrant
211	Top to bottom 180 degrees
212	Right to left 180 degrees

Value	Description
213	Top to bottom 90 degrees
214	Right to left 90 degrees
221	Top 180 degrees
222	Right 180 degrees
223	Bottom 180 degrees
224	Left 180 degrees
225	Counter rotating top bottom
226	Counter rotating left right
227	Double rotating top bottom
228	Double rotating left right
231	V Open top
232	V Open right
233	V Open bottom
234	V Open left
235	V Open top bottom
236	V Open left right
241	Rotating top left
242	Rotating bottom left
243	Rotating bottom right
244	Rotating top right
245	Rotating top left bottom right
246	Rotating bottom left top right
251	Rotating top left right
252	Rotating left top bottom
253	Rotating bottom left right
254	Rotating right top bottom
261	Rotating double center right

Value	Description
262	Rotating double center top
263	Rotating double center top bottom
264	Rotating double center left right
409	Random effect (One of the 133 SMPTE effects is chosen at random)
503	Random radial (One of the 39 SMPTE radial effects is chosen at random)

SMPTE Matrix Effects

`kMatrixTransitionType ('smp4')`

This effect is an implementation of the 34 matrix effects from ANSI/SMPTE 258M-1993, plus two Apple-defined wipes that choose a random effect. These are a series of matrix “reveal” type effects that take place between two sources. For full definitions of these 34 matrix effects and what they look like, refer to the SMPTE documentation.

The SMPTE matrix effects take two sources and seven parameters.

Use the descriptions below to help you understand what the parameters do. To learn how to use parameter atoms, see [Adding Video Effects to a QuickTime Movie](#) (page 11).

Name	Code	QTAtom Type	Description
Percentage	'pcnt'	kParameterType-DataFixed; Always a tween	This parameter contains a tween. As the effect progresses, QuickTime renders the frame of the effect indicated by the tween's current value, as a percentage of the whole effect. For example, if the tween goes from 0 to 100, the effect renders completely; if the tween goes from 25 to 75, rendering begins 25% into the effect and terminates 75% through the effect.
Wipe ID	'wpID'	kParameterType-DataEnum	Contains the SMPTE ID for the effect. By setting this parameter, you control which of the 34 available matrix effects is used. See the enumeration list below.
Soft border	'soft'	kParameterType-DataBitField	If this parameter contains <code>true</code> , the border drawn around the second source is blurred.
Border width	'widt'	kParameterType-DataFixed; Can be a tween	The width, in pixels, of a border that is drawn around the second source.
Border color	'bc1r'	kParameterType-DataRGBValue; Can be a tween	The RGB color of the border around the second source.

Name	Code	QTAtom Type	Description
Horizontal repeat	'hori'	kParameterType-DataLong; Can be a tween	The number of horizontal repeats of the effect that are executed in a single source.
Vertical repeat	'vert'	kParameterType-DataLong; Can be a tween	The number of vertical repeats of the effect that are executed in a single source.

The Wipe ID parameter can take the following values:

Value	Description
301	Horizontal matrix
302	Vertical matrix
303	Top left diagonal matrix
304	Top right diagonal matrix
305	Bottom right diagonal matrix
306	Bottom left diagonal matrix
310	Clockwise top left matrix
311	Clockwise top right matrix
312	Clockwise bottom right matrix
313	Clockwise bottom left matrix
314	Counter clockwise top left matrix
315	Counter clockwise top right matrix
316	Counter clockwise bottom right matrix
317	Counter clockwise bottom left matrix
320	Vertical start top matrix
321	Vertical start bottom matrix
322	Vertical start top opposite matrix
323	Vertical start bottom opposite matrix
324	Horizontal start left matrix
325	Horizontal start right matrix
326	Horizontal start left opposite matrix
327	Horizontal start right opposite matrix

Value	Description
328	Double diagonal top right matrix
329	Double diagonal bottom right matrix
340	Double spiral Top matrix
341	Double spiral bottom matrix
342	Double spiral left matrix
343	Double spiral right matrix
344	Quad spiral vertical matrix
345	Quad spiral horizontal matrix
350	Vertical waterfall left matrix
351	Vertical waterfall right matrix
352	Horizontal waterfall left matrix
353	Horizontal waterfall right matrix
409	Random effect (One of the 133 SMPTE effects is chosen at random)
504	Random matrix (One of the 34 SMPTE matrix effects is chosen at random)

Video Effects from Apple

The following video effects are supplied by Apple Computer and are built into QuickTime.

Alpha Compositor

```
kAlphaCompositorTransitionType ('blnd')
```

This effect is used to combine two images using the alpha channels of the images to control the blending. It provides for the standard alpha blending options, and can handle pre-multiplying by any color, although white and black are most common and often run faster.

The alpha compositor effect takes a maximum of two sources and has two parameters.

Use the descriptions below to help you understand what the parameters do. To learn how to use parameter atoms, see [Adding Video Effects to a QuickTime Movie](#) (page 11).

Name	Code	QTAtom Type	Description
Blend mode	'bMod'	kParameterType-DataEnum	Contains the blend mode for the effect. See the enumeration list below.
Pre-multiply color	'mclr'	kParameterType-DataRGBValue	If the blend mode is "pre-multiply alpha," this parameter contains the color used in the pre-multiply blend, otherwise it is ignored.

Blend Mode Enum

The blend mode parameter can contain one of the following values:

- **Straight alpha:** perform a standard alpha blend. The alpha channel value of the first source defines the amount of the first source that is included in the composited image, and one minus the alpha channel value of the first source defines the amount of the second source that is included in the composited image.
- **Pre-multiply alpha:** calculates the destination pixel according to the following formulae:

```
DestinationRed = PreMultiplyRed * (1-alphaC) + temp1 * alphaC
DestinationGreen = PreMultiplyGreen * (1-alphaC) + temp2 * alphaC
DestinationBlue = PreMultiplyBlue * (1-alphaC) + temp3 * alphaC
```

where:

```
alphaC = alphaB + (1-alphaB) * alphaA
temp1 = (alphaA * SourceARed + alphaB * sourceBRed)/alphaC
temp2 = (alphaA * SourceAGreen + alphaB * sourceBGreen)/alphaC
temp3 = (alphaA * SourceABlue + alphaB * sourceBBlue)/alphaC
```

- **Reverse alpha:** perform a reverse alpha blend. The one minus the alpha channel value of the first source defines the amount of the first source that is included in the composited image, and the alpha channel value of the first source defines the amount of the second source that is included in the composited image.

Alpha Gain filter

```
kAlphaGainImageFilterType ('gain')
```

The alpha gain filter is used to alter the alpha channel of a single source. This operation is commonly applied before passing the source to the alpha compositor effect described above. The following equation describes the alteration that is made to the source's alpha channel:

```
newAlpha = bottomPin <= (gain*oldAlpha + offset) <= topPin
```

This means that to increase the alpha channel by a set amount, you set the gain parameter to 1.0, and the offset to the desired increase. Similarly, to increase the alpha channel by a fixed percentage, set the offset to 0.0 and the gain to the percentage increase desired. The `topPin` and `bottomPin` parameters allow you to set upper and lower bounds on the value of the alpha channel, respectively.

The alpha gain filter effect takes a maximum of one source and has four parameters.

Use the descriptions below to help you understand what the parameters do. To learn how to use parameter atoms, see [Adding Video Effects to a QuickTime Movie](#) (page 11).

Name	Code	QTAtom Type	Description
Gain value	'gain'	kParameterType-DataFixed	This value is multiplied by the original alpha channel value.
Offset value	'offs'	kParameterType-DataFixed	This value is added to the old alpha channel, after it has been multiplied by the gain parameter.
Top alpha pin	'pinT'	kParameterType-DataFixed	The maximum value that the alpha channel can take after the gain and offset parameters have been applied.
Bottom alpha pin	'pinB'	kParameterType-DataFixed	The minimum value that the alpha channel can take after the gain and offset parameters have been applied.

Blur Filter

```
kBlurImageFilterType ('blur')
```

This effect applies a convolution blur effect to a single source. The actual blur that is applied is determined by the convolution kernel. This is a matrix of values that are applied to each pixels of the source to produce the destination.

The Blur effect takes a maximum of one source and has two parameters.

Use the descriptions below to help you understand what the parameters do. To learn how to use parameter atoms, see [Adding Video Effects to a QuickTime Movie](#) (page 11).

Name	Code	QTAtom Type	Description
Amount of blurring	'ksiz'	kParameterType-DataEnum	The size of the blur kernel to apply. This value must be one of 3, 5, 7, 9, 11, 13 or 15. The larger the kernel, the longer the effect will take to run and the greater the degree of blurring.
Brightness	'ksum'	kParameterType-DataFixed	This is the total value of the elements of the blur kernel. Normally this value will be 1.0, which blurs the source but doesn't change its brightness. If the value is between 0.0 and 1.0, the brightness is decreased, if the value is greater than 1.0, the brightness is increased.

Chroma Key

`kChromaKeyTransitionType ('ckey')`

The chroma key effect combines two sources by replacing all the pixels of the first source that are the specified color with the corresponding pixels of the second source. This allows the second source to “show through” the first in those places where the first source is the given color.

This has the effect of putting the second source “behind” the first source, and making the selected color “transparent”.

The chroma key effect takes a maximum of two sources and has one parameter.

Use the descriptions below to help you understand what the parameters do. To learn how to use parameter atoms, see [Adding Video Effects to a QuickTime Movie](#) (page 11).

Name	Code	QTAtom Type	Description
Key color	'keyc'	kParameterTypeDataRGBValue	Pixels of this color in the first source will be replaced by pixels from the second source.

Cloud

`kCloudCodecType ('clou')`

The cloud effect uses a fractal noise generator to simulate a cloud formation. This can be transparently overlaid on an image. The cloud formation’s colors can be controlled, and the cloud randomly changes shape over time.

The cloud effect takes no sources and has two parameters.

Use the descriptions below to help you understand what the parameters do. To learn how to use parameter atoms, see [Adding Video Effects to a QuickTime Movie](#) (page 11).

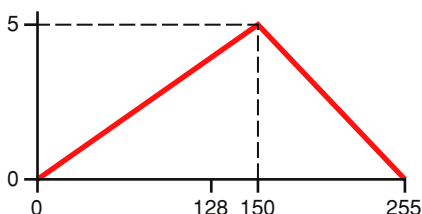
Name	Code	QTAtom Type	Description
Cloud color	'fgdc'	kParameterTypeDataRGBValue; Can be a tween	The foreground color of the cloud.
Background color	'bckc'	kParameterTypeDataRGBValue; Can be a tween	The background color of the cloud.
Rotation	'rotc'	kParameterTypeDataDouble; Can be a tween	The vertical rotation of the cloud around its axis. Legal range is 0 to 360.

Color Style

```
kSolarizeImageFilterType ('solr')
```

The color style effect allows you to apply two color stylizations to a single source. They are:

- **Solarization:** adjusts the color balance of the source by constructing a table of replacement color values from two parameters. These parameters are the maximum color intensity and the peak point of the color spread. The table starts at zero intensity and increases to the maximum intensity at the peak point. After that it falls back to zero. For example, if the color values range from 0 to 255, the maximum intensity is 5 and the peak point is at 150, the resulting table's profile will look like Figure 3-1.



- **Posterization:** reduces the number of colors in an image by replacing all pixels whose color is in a consecutive range with the middle color from that range. This produces a “color banding” effect.

Both these effects work on a per-channel basis, which means that the red, green and blue components of each pixel are independently passed through the respective algorithm.

For solarization, a maximum intensity of 1 and a peak point of 128 are the most commonly used values.

The color style effect takes a maximum of one source and has three parameters.

Name	Code	QTAtom Type	Description
Solarize amount	'solr'	kParameterTypeDataLong; Can be a tween	The maximum intensity of the solarization table.
Solarize point	'solp'	kParameterTypeDataLong; Can be a tween	The peak point of the solarization table.
Posterize amount	'post'	kParameterTypeDataLong; Can be a tween	The number of colors that are grouped and replaced with the mid-range color.

ColorSync filter

```
kColorSyncImageFilterType ('sync')
```

The color sync filter adjusts the color balance of an image to match a specified color sync profile. Typically, you would use this to adjust the color profile of an image to match the current display device. This allows you to maintain accurate color representations across devices. You specify both the color sync profile of the source image and the color sync profile of the destination device the image will be rendered to.

The color sync filter takes a maximum of one source and has two parameters.

Use the descriptions below to help you understand what the parameters do. To learn how to use parameter atoms, see [Adding Video Effects to a QuickTime Movie](#) (page 11).

Name	Code	QTAtom Type	Description
Source profile	'srcP'	kParameterTypeDataEnum	The color sync profile of the source image.
Destination profile	'destP'	kParameterTypeDataEnum	The color sync profile of the target device.

Color Tint filter

`kColorTintImageFilterType ('tint')`

The color tint filter converts its source into greyscale, then applies a light and a dark color to the image. The light color replaces the white in the greyscale image, and the dark color replaces the black. This filter also includes brightness and contrast controls. The end result is a tinted duochrome version of the source image.

You can use this filter, for example, to apply a sepia tone to a source.

The color tint filter takes a maximum of one source and has four parameters

Use the descriptions below to help you understand what the parameters do. To learn how to use parameter atoms, see [Adding Video Effects to a QuickTime Movie](#) (page 11).

Name	Code	QTAtom Type	Description
Tint	'tint'	kParameterTypeDataEnum	1 = Black and White; 2 = X-Ray; 3 = Sepia; 4 = Cobalt; 0 = dividing line for pop up; 100 = custom, in which case the other parameters take effect.
Dark color	'back'	kParameterType-DataRGBValue; Can be a tween	The color to use to replace the black of the greyscale image.
Light color	'fore'	kParameterType-DataRGBValue; Can be a tween	The color to use to replace the white of the greyscale image.
Brightness	'brig'	kParameterTypeDataLong; Can be a tween	The amount to adjust the brightness of the source by, ranging from -255 (all colors are replaced with black) to 255 (all colors are replaced with white). A value of 0 will leave the brightness unchanged.

Name	Code	QTAtom Type	Description
Contrast	'cont'	kParameterTypeDataLong; Can be a tween	The amount to adjust the contrast of the source by, ranging from -255 (minimum contrast) to 255 (maximum contrast). A value of 0 will leave the contrast unchanged.

Edge Detection Filter

`kEdgeDetectImageFilterType ('edge')`

This effect applies an edge detection convolution to a single source. The performance of the edge detection is determined by the convolution kernel. This is a matrix of values applied to each pixel of the source to produce the resulting image.

This effect takes a maximum of one source, and has two parameters.

Use the descriptions below to help you understand what the parameters do. To learn how to use parameter atoms, see [Adding Video Effects to a QuickTime Movie](#) (page 11).

Name	Code	QTAtom Type	Description
Edge thickness	'ksiz'	kParameterType-DataEnum	The size of the kernel to apply. This value must be one of 3, 5, 7, 9, 11, 13 or 15. Larger kernels will produce thicker edges in the resulting image.
Colorize	'colz'	kParameterType-DataBitField	If this parameter is set to <code>true</code> , the color of the edges produced by the effect are based on the color of the source pixels around them. Otherwise, edges are rendered as light grey against a dark grey background.

Emboss Filter

`kEmbossImageFilterType ('embs')`

This effect applies an emboss convolution to a single source. The performance of the embossing operation is determined by the convolution kernel. This is a matrix of values applied to each pixel of the source to produce the resulting image.

This effect takes a maximum of one source, and has one parameter, amount of embossing.

Use the descriptions below to help you understand what the parameters do. To learn how to use parameter atoms, see [Adding Video Effects to a QuickTime Movie](#) (page 11).

Name	Code	QTAtom Type	Description
Amount of embossing	'ksiz'	kParameterType-DataEnum	The size of the kernel to apply. This value must be one of 3, 5, 7, 9, 11, 13 or 15. Larger kernels will produce a more heavily embossed result.

Explode

`kExplodeTransitionType('xplo')`

In an explode effect, source B grows from a single point, expanding out until it entirely covers source A. The center point of the explosion is defined in the effect parameters.

This effect takes a maximum of two sources, and has three parameters.

Use the descriptions below to help you understand what the parameters do. To learn how to use parameter atoms, see [Adding Video Effects to a QuickTime Movie](#) (page 11).

Name	Code	QTAtom Type	Description
Percentage	'pcnt'	kParameterType-DataFixed; Always a tween	This parameter contains a tween. As the effect progresses, QuickTime renders the frame of the effect indicated by the tween's current value, as a percentage of the whole effect. For example, if the tween goes from 0 to 100, the effect renders completely; if the tween goes from 25 to 75, rendering begins 25% into the effect and terminates 75% through the effect.
Explode centre X	'xcnt'	kParameterType-DataFixed; Can be a tween	The x-coordinate of the explosion centre.
Explode centre Y	'ycnt'	kParameterType-DataFixed; Can be a tween	The y-coordinate of the explosion centre.

Film Noise Filter

`kFilmNoiseImageFilterType('fmns')`

The film noise filter alters a single source, simulating some of the effects that are seen on aged film stock. This effect can be used to transform a video source into one that looks like it was shot on film that has suffered the effects of age and mishandling.

The specific features, which can be controlled independently, are:

- Hairs. These are a simulation of hairs lying on the surface of the film. Each hair is randomly generated, and is colored in a randomly chosen shade of light grey.

- **Scratches.** These are vertical or near-vertical one-pixel lines drawn onto the destination image that simulate scratches in the film. Each scratch lasts for a pre-calculated length of time. During its lifespan the scratch's position is randomly perturbed. Shortly before the scratch is removed, it will begin to shorten. The color of the scratches is a randomly chosen shade of light grey.
- **Dust.** These simulate dust particles on the surface of the film. Dust particles are drawn using the same algorithm that generates the hairs, but the particles are more tightly curled, and drawn in a darker shade of grey.
- **Film fade.** This simulates an overall change in the color of the film stock. Every pixel of the source image is passed through the film fade algorithm, so this can be processor-intensive.

The film noise effect takes a single source and has eight parameters.

Use the descriptions below to help you understand what the parameters do. To learn how to use parameter atoms, see [Adding Video Effects to a QuickTime Movie](#) (page 11).

Name	Code	QTAtom Type	Description
Hair density	'hden'	kParameterType-DataLong	This parameter controls the number of hairs that are drawn on each frame and the frequency with which hairs appear. The maximum number of hairs per frame is a randomly generated number between 1 and the value of this parameter. The chance of each hair appearing on a single frame is 1 in (the value of this parameter).
Hair length	'hlen'	kParameterType-DataLong	The maximum length (in pixels) of the hairs being drawn.
Scratch density	'sden'	kParameterType-DataLong	This parameter controls the number of scratches that are drawn on each frame and the frequency with which scratches appear. The maximum number of scratches per frame is a randomly generated number between 1 and the value of this parameter. The chance of each scratch appearing on a single frame is 1 in (the value of this parameter).
Scratch duration	'sdur'	kParameterType-DataLong	The maximum number of frames that each scratch appears for. The actual number of frames for each scratch is a randomly chosen value between 1 and this value plus 20.
Scratch width	'swid'	kParameterType-DataLong	The maximum width, in pixels, of a scratch. The actual width is a randomly chosen number between 1 and this value.
Dust density	'dden'	kParameterType-DataLong	This parameter controls the number of dust particles that are drawn on each frame and the frequency with which dust particles appear. The maximum number of dust particles per frame is a randomly generated number between 1 and the value of this parameter. The chance of each dust particle appearing on a single frame is 1 in (the value of this parameter).

Name	Code	QTAtom Type	Description
Dust size	'dsiz'	kParameterType-DataLong	For each dust particle, the length in pixels is a random number between 1 and 5, plus the value of the Dust Size parameter.
Film fade	'fade'	kParameterType-DataEnum	The type of film fade effect (if any) to apply. See list below.

The Film Fade Enum

The film fade parameter can take one of the following values:

Value	String	Description
1	None	The destination image is a copy of the source.
2	Sepia tone	The destination is a slightly saturated, monochromatic version of the source, colorized into shades of light red-brown.
3	Black and white	The destination is a greyscale version of the source.
4	Faded color film	The destination is a color desaturated version of the source.
5	1930's color film	The destination is a color supersaturated version of the source.

Fire

`kFireCodecType ('Fire')`

The fire effect simulates a fire by generating a number of individual flames of randomized appearance. You can control various parameters that define how the flames are generated and how they change over time.

The fire effect takes no sources and has four parameters.

Use the descriptions below to help you understand what the parameters do. To learn how to use parameter atoms, see [Adding Video Effects to a QuickTime Movie](#) (page 11).

Name	Code	QTAtom Type	Description
Spread rate	'sprd'	kParameterType-DataLong	How quickly the fire expands to its highest level from its starting point. The higher the value, the more quickly the fire starts up and reaches its maximum burn rate.
Sputter rate	'decy'	kParameterType-DataLong	How quickly the flames die down as they move up the screen. Low numbers result in very tall flames, high numbers in very low flames.

Name	Code	QTAtom Type	Description
Water rate	'watr'	kParameterType-DataLong	How often "water" is tossed on the base of the fire, instantly putting out the fire at that point. High numbers result in a fire that's very broken up (i.e. many areas of burning and non-burning) while lower numbers result in a wider, smoother fire.
Restart rate	'rset'	kParameterType-DataLong	How often entire fire is put out, then allowed to restart.

General Convolution Filter

`kConvolveImageFilterType ('genk')`

This effect applies a general purpose convolution effect to a single source. The effect that results is completely determined by the values entered into the kernel parameters of the effect. The kernel for this convolution is always a 3-by-3 matrix of values.

The values of the cells of the convolution kernel determine the value that is assigned to each pixel of the destination frame. The convolution algorithm examines every pixel of the source, and the eight pixels surrounding it. These values are multiplied by the appropriate values in the cells and summed. This sum is then used as the value of the corresponding destination pixel.

Many computer graphics textbooks offer a more complete and rigorous explanation of convolution, and you are encouraged to consult these for useful values that the kernel can take. A great introduction to convolution for programmers can be found in *High Performance Computer Imaging*, Chapter 6, by Ihtisham Kabir, Manning Publications, 1996. ISBN 1-884777-26-0. A somewhat more classical reference is *Computer Graphics: Principles and Practice, 2nd. Edition*, pp. 629-636, Foley J.D., van Dam A., Feiner S.K. and Hughes J.F., Addison-Wesley, 1990. ISBN 0-201-12110-7.

As an example of a kernel that produces a useful result, the kernel values shown in Figure 3-2 will shift an image left by one pixel.

$$\begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

The General Convolution Filter effect takes a maximum of one sources and has nine parameters.

Use the descriptions below to help you understand what the parameters do. To learn how to use parameter atoms, see [Adding Video Effects to a QuickTime Movie](#) (page 11).

Name	Code	QTAtom Type	Description
Cell one	'ce11'	kParameterTypeDataFixed	The value to be placed into the first cell of the kernel.
Cell two	'ce12'	kParameterTypeDataFixed	The value to be placed into the second cell of the kernel.

Name	Code	QTAtom Type	Description
Cell three	'ce13'	kParameterTypeDataFixed	The value to be placed into the third cell of the kernel.
Cell four	'ce14'	kParameterTypeDataFixed	The value to be placed into the fourth cell of the kernel.
Cell five	'ce15'	kParameterTypeDataFixed	The value to be placed into the fifth cell of the kernel.
Cell six	'ce16'	kParameterTypeDataFixed	The value to be placed into the sixth cell of the kernel.
Cell seven	'ce17'	kParameterTypeDataFixed	The value to be placed into the seventh cell of the kernel.
Cell eight	'ce18'	kParameterTypeDataFixed	The value to be placed into the eighth cell of the kernel.
Cell nine	'ce19'	kParameterTypeDataFixed	The value to be placed into the ninth cell of the kernel.

The nine cells in the kernel are laid out as shown in Figure 3-3.

Cell one	Cell two	Cell three
Cell four	Cell five	Cell six
Cell seven	Cell eight	Cell nine

Gradient Wipe

```
kGradientTransitionType ('matt')
```

The gradient wipe effect uses a matte image to create a transition between two source images. The transition from source 'A' to source 'B' will occur first where the matte image is darkest, last where the matte image is brightest.

During the effect, if the luminance value of the matte image at a given point is greater than the alpha threshold for the effect, the pixel from source 'A' is displayed. If the matte image's luminance value at that point is less than the alpha threshold, the pixel from source 'B' is displayed.

The alpha threshold increases as the effect progresses, eventually causing it to be higher than the luminance value of the matte image at all points, so that only the pixels from source 'B' are displayed.

The equation for the change in the alpha threshold as the effect progresses is:

$$\text{alphaThreshold} = (\text{percent_complete} / 100) * 255$$

so the alpha threshold goes from 0 to 255 over the course of the effect.

The algorithm used to animate the transition is:

```
for (y=0; y<height; y++) {
  for (x=0; x<width; x++) {
    if (matte_image_luminance(x,y) > alphaThreshold) {
      output pixel source_A(x,y);
    } else {
      output pixel source_B(x,y);
    }
  }
}
```

The gradient wipe effect takes two sources and has two parameters.

Use the descriptions below to help you understand what the parameters do. To learn how to use parameter atoms, see [Adding Video Effects to a QuickTime Movie](#) (page 11).

Name	Code	QTAtom Type	Description
Percentage	'pcnt'	kParameterType-DataFixed; Always a tween	This parameter contains a tween. As the effect progresses, QuickTime renders the frame of the effect indicated by the tween's current value, as a percentage of the whole effect. For example, if the tween goes from 0 to 100, the effect renders completely; if the tween goes from 25 to 75, rendering begins 25% into the effect and terminates 75% through the effect.
Matte image	'matt'	kParameterType-DataImage	The matte image. The transition from source 'A' to source 'B' will occur first where the matte image is darkest, last where the matte image is brightest. A greyscale matte image is recommended.

HSL Balance Filter

```
kHSLColorBalanceImageFilterType ('hsvb')
```

This filter effect allows you to independently adjust the hue, saturation and lightness (also known as value or brightness) channels of a single source. The effect adjusts every pixel in the source, multiplying the hue component of the pixel by the value of the hue multiplier parameter, the saturation component by the value of the saturation multiplier parameter, and so on.

The HSL balance filter effect takes one source and has three parameters.

Use the descriptions below to help you understand what the parameters do. To learn how to use parameter atoms, see [Adding Video Effects to a QuickTime Movie](#) (page 11).

Name	Code	QTAtom Type	Description
Hue multiplier	'hmul'	kParameterTypeDataFixed; Can be a tween	The amount by which to adjust the hue channel value of each pixel.

Name	Code	QTAtom Type	Description
Saturation multiplier	'smul'	kParameterTypeDataFixed; Can be a tween	The amount by which to adjust the saturation channel value of each pixel.
Lightness multiplier	'vmul'	kParameterTypeDataFixed; Can be a tween	The amount by which to adjust the lightness channel value of each pixel.

Implode

`kImplodeTransitionType('mplo')`

In an implode effect, source A shrinks down to a single point, revealing source B. The center point of the implosion is defined in the effect parameters.

The implode effect takes a maximum of two sources and has three parameters.

Use the descriptions below to help you understand what the parameters do. To learn how to use parameter atoms, see [Adding Video Effects to a QuickTime Movie](#) (page 11).

Name	Code	QTAtom Type	Description
Percentage	'pcnt'	kParameterType-DataFixed; Always a tween	This parameter contains a tween. As the effect progresses, QuickTime renders the frame of the effect indicated by the tween's current value, as a percentage of the whole effect. For example, if the tween goes from 0 to 100, the effect renders completely; if the tween goes from 25 to 75, rendering begins 25% into the effect and terminates 75% through the effect.
Implode centre X	'xcnt'	kParameterType-DataFixed; Can be a tween	The x-coordinate of the implosion centre.
Implode centre Y	'ycnt'	kParameterType-DataFixed; Can be a tween	The y-coordinate of the implosion centre.

Lens Flare

`kLensFlareImageFilterType('lens')`

The lens flare effect simulates the effect of light reflecting from a camera lens. You can select the shape of the flare from a list, set the size and brightness of the flare effect, and set x and y values that will cause the flare to move across the lens during the effect. You can also set several parameters that affect specific aspects of the flare, such as the number of sides for a polygon flare or the flare color.

The lens flare effect takes one source and has fifteen parameters.

Use the descriptions below to help you understand what the parameters do. To learn how to use parameter atoms, see [Adding Video Effects to a QuickTime Movie](#) (page 11).

Name	Code	QTAtom Type	Description
Size	'size'	kParameterTypeDataFixed; Can be a tween	Size of the effect Range 1-20. Default 1.
Brightness	'gbri'	kParameterTypeDataFixed; Can be a tween	Brightness of the effect Range 0-1. Default 1.
X	'xcnt'	kParameterTypeDataFixed; Always a tween	X-coordinate of the effect center Range -2 to 2. Default 0.
Y	'ycnt'	kParameterTypeDataFixed; Always a tween	Y-coordinate of the effect center Range -2 to 2. Default 0.
Type	'ftyp'	kParameterEnumList	Type of flare See enumeration list below.
Color	'flrc'	kParameterTypeDataRGBValue	Flare color Black or white. Default white.
Brightness	'fbri'	kParameterTypeDataFixed	Flare brightness Range 0-2. Default 1.
Size	'fsiz'	kParameterTypeDataFixed	Flare size Range 0-1. Default 0.5.
Position	'fylc'	kParameterTypeDataFixed	Position of flare element along centerline Range -2 to 2. Default 0.
Offset	'fxlc'	kParameterTypeDataFixed	Offset of flare element's center (creates a hole if nonzero) Range -2 to 2. Default 0.
Solid	'fsz1'	kParameterTypeDataFixed	Portion of flare that doesn't fade Range -2 to 2. Default 0.
Anamorphic	'fana'	kParameterTypeDataBitField	Flare is round if false, oval if true. Default false.
Number	'fnum'	kParameterTypeDataLong	The number of spikes (for a starburst) or sides (for a polygon) Range 1-20. Default 1.
Count	'fcnt'	kParameterTypeDataLong	The number of flare elements. If Count = N > 1, there are a random number of flare elements (1 to N). Range 1-50. Default 1.
Seed	'fsed'	kParameterTypeDataLong	Seed for the random number generator if Count > 1. Range 0-1000. Default 500.

The following table list spot values.

Value	String	Description
1	Spot	A linear spot

Value	String	Description
2	Round Spot	A squared-distance spot
3	Reverse Spot	A linear spot, center dark rather than light
4	Reverse Round Spot	A squared-distance spot, center dark rather than light
5	Star Burst	A spikey ball with 'Number' spikes
6	Polygon	A polygon with 'Number' sides

RGB Balance Filter

`kRGBColorBalanceImageFilterType ('rgbb')`

The RGB balance filter allows you to independently adjust the red, green, blue, and alpha channels of a single source. The effect adjusts every pixel in the source, multiplying the red component of the pixel by the value of the red multiplier parameter, the green component by the value of the green multiplier parameter, and so on.

The RGB balance filter takes a maximum of one source and has three parameters.

Use the descriptions below to help you understand what the parameters do. To learn how to use parameter atoms, see [Adding Video Effects to a QuickTime Movie](#) (page 11).

Name	Code	QTAtom Type	Description
Red multiplier	'rml'	kParameterTypeDataFixed; Can be a tween	The amount to adjust the red channel value of each pixel by.
Green multiplier	'gml'	kParameterTypeDataFixed; Can be a tween	The amount to adjust the green channel value of each pixel by.
Blue multiplier	'bml'	kParameterTypeDataFixed; Can be a tween	The amount to adjust the blue channel value of each pixel by.

Ripple

`kWaterRippleCodecType ('ripl')`

The ripple effect simulates a pool of water that overlays an image. The area within the ripple mask will undulate, giving the appearance of water. If the user clicks within the ripple area, concentric waves are sent across the water, simulating a stone dropped into the pool.

The ripple effect takes no sources and has one parameter, ripple mask.

Use the descriptions below to help you understand what the parameters do. To learn how to use parameter atoms, see [Adding Video Effects to a QuickTime Movie](#) (page 11).

Name	Code	QTAtom Type	Description
Ripple mask	'mask'	kParameterType-DataImage	A 1-bit image that acts as a mask; the ripple effect is seen at every pixel corresponding to a pixel in the mask that is set.

Sharpen Filter

`kSharpenImageFilterType ('shrp')`

This effect applies a convolution sharpen effect to a single source. The sharpening that is applied is determined by the convolution kernel. This is a matrix of values that are applied to each pixel of the source to produce the destination.

The sharpen filter effect takes one source and has two parameters.

Use the descriptions below to help you understand what the parameters do.

Name	Code	QTAtom Type	Description
Amount of sharpening	'ksiz'	kParameterType-DataEnum	The size of the sharpen kernel to apply. This value must be one of 3, 5, 7, 9, 11, 13 or 15. The smaller the kernel, the faster the effect will run and the greater the degree of sharpening.
Brightness	'ksum'	kParameterType-DataFixed	This is the total value of the elements of the sharpen kernel. Normally this value will be 1.0, which sharpens the source but doesn't change its brightness. If the value is between 0.0 and 1.0, the brightness is decreased, if the value is greater than 1.0, the brightness is increased.

To learn how to use parameter atoms, see [Adding Video Effects to a QuickTime Movie](#) (page 11).

Creating New Video Effects

This chapter discusses how to write your own video effects. If you are only interested in building applications that use effects, you can skip this chapter.

QuickTime video effects are implemented as Component Manager components, the standard mechanism for extending QuickTime. To implement your own effect, you create a new **effect component**. An effect component is a specialized type of image decompressor component.

This chapter walks you through the implementation of a sample effect component. The sample effect is built on a framework of code that you can reuse when you implement your own effect component.

What Effects Components Do

The basic task of every effect component is very simple. The component is passed zero or more source frames and must produce a single destination frame. The destination frame is the source frame or frames after processing by the effect-rendering algorithm.

The component must provide a set of services that QuickTime can call. These services allow QuickTime (or any other client software that uses your component) to perform actions such as these:

- Open a connection to your component.
- Retrieve information about your effect, particularly descriptions of the parameters your effect can take.
- Set the source or sources for the effect.
- Set the destination for the effect.
- Request that a single frame of the effect be rendered.
- Cancel the rendering of a frame.
- Close the connection to your component.

Your effect component must be able to service such requests. To do so, it implements a set of standard **interface functions** that are called through a **component dispatch** function. Details of these functions are given in the section [The Effect Component Interface](#) (page 60).

The main task of the effect component is to implement the specific algorithm that transforms source frames into a destination frame. You need to supply versions of your algorithm for each bit depth and pixel format that your component supports. Choosing which bit depths and pixel formats to support, and implementing algorithms for each combination of these, are a significant part of building your effect component.

In addition, your effect component must provide a **parameter description** that describes the parameters that the effect takes. The parameter description can be used by the software that is calling your component to construct a user interface that allows users to change the value of the parameters sent to your component. This is described in detail in the section [Supplying Parameter Description Information](#) (page 61).

The Effect Component Interface

Effect components, as with all other types of QuickTime components, must implement a defined set of functions. To ease the component development process, the Generic Effect component is provided for you. This component implements many of the “housekeeping” functions that all components must perform. In most cases, these default implementations are appropriate for your effect, and you simply delegate these functions to the generic effect component. In the rare instances when you need to provide your own implementations of one of these basic functions, you can override the generic version and provide your own implementation.

By delegating many of the functions to the generic effect, you not only decrease the number of functions you must implement, you also produce a smaller effect component, because common code is stored only once, in the generic effect.

The framework code provided in the dimmer effect sample ([The Dimmer Effect](#) (page 71)), shows how to delegate interface functions to the generic effect component.

Your component must provide implementations for these functions:

Term	Definition
Open	Opens a connection between the client software and your component.
Close	Closes the connection between the client and your component.
Version	Returns the version number of your component.
EffectSetup	Called once before a sequence of frames are rendered. This gives your effect the chance to set up variables that will alter their value during the execution of a sequence of frames.
EffectBegin	Called once before a frame is rendered. Your component can safely perform operations that move memory when this function is called.
EffectRenderFrame	Called to render a frame. Because this function can be called asynchronously, it is not safe to perform operations that may move memory during this call.
EffectCancel	Cancels the rendering of a frame. If your component supports asynchronous operation, this function can be called while a frame is being rendered.
GetParameter-ListHandle	Returns a parameter description atom container, as described in the section “Supplying Parameter Description Information” [link s Creating New Video Effects].
GetCodecInfo	Returns information to the codec manager about the capabilities of your component.
EffectGetSpeed	Returns the approximate number of frames per second that your effect is capable of transforming.

These functions can be categorized into four groups. The `Open` and `Close` functions deal with maintaining a connection between your component and client software. In most cases, you can implementation these functions using the sample code provided by Apple without modification.

The `Version`, `GetParameterListHandle`, `GetCodecInfo` and `EffectGetSpeed` functions return information about your component. The most important of these functions is `GetParameterListHandle`, which returns a description of the parameters that your effect can take. See [Supplying Parameter Description Information](#) (page 61) for more details of this what this function should do.

The `EffectSetup` function is called immediately before your component is required to render a sequence of frames. On entry, the function contains a description of the sequence that is about to be rendered. Most importantly, it describes the bit depth and pixel format of the sources that your component has to deal with. Your `Setup` function can then verify that your component can handle these formats. If it cannot, `EffectSetup` should return the “closest” bit depth and pixel format combination that it can handle, and QuickTime will generate versions of the sources and destination in the requested format. This ensures that your effect component is given source and destination buffers in a format that it understands. See [The EffectRenderFrame Function](#) (page 67) for more details.

The most significant function group contains the `EffectBegin`, `EffectRenderFrame`, and `EffectCancel` functions. These functions contain the implementation of your effect algorithm. In most cases, you can implement the `EffectCancel` function simply by using the sample code provided by Apple. The implementation of the `EffectBegin` and `RenderFrame` functions is covered in [Implementing the EffectBegin and EffectRenderFrame Functions](#) (page 62).

Full details of the interface functions your component must supply are given in [Component-Defined Functions](#) (page 91).

Supplying Parameter Description Information

Your effect component must supply information that describes the parameters that your effect takes. This information is used to create an appropriate user interface for setting the parameters to your effect. The parameter description lists your effect’s parameters and their data types and indicates the appropriate selection interface for each parameter, such as a slider or a pull-down list, as well as information such as the minimum, maximum, and default values for each parameter. Each parameter is described using a specific format, which is shown in [The Parameter Description Format](#) (page 78).

Your effect component returns its parameter description information through the `GetParameterListHandle` function. The easiest way to provide this information back to the client software is to add an `'atms'` resource to your component. The `'atms'` resource contains the parameter descriptions in the required format. You can then retrieve the resource by calling the `GetComponentResource` function, returning it to the client through your implementation of `GetParameterListHandle`, as shown in Listing 4-1.

Listing 4-1 Implementing the `GetParameterListHandle` function using `GetComponentResource`

```
pascal ComponentResult GetParameterListHandle(EffectGlobals *glob,
                                             Handle *theHandle)
{
    OSErr    err = noErr;
    err = GetComponentResource((Component) glob->self,
                              OSTypeConst('atms'),
                              kEffectatmsRes,
                              theHandle);
    return err;
}
```

By implementing the `GetParameterListHandle` function in this way, you can simplify the process of packaging the necessary information in the proper format.

Implementing the `EffectBegin` and `EffectRenderFrame` Functions

The core of implementing an effect component is implementing the `EffectBegin` and `EffectRenderFrame` functions. Together, these functions handle the rendering of a single frame of the effect.

The `EffectBegin` function is called immediately before each frame is to be rendered. It is guaranteed that this function is never called from an interrupt, so it is safe to perform actions that could move memory within this function. In general, the `EffectBegin` function should set up the internal state of your component so it has all the information it needs to render a single frame.

The `EffectRenderFrame` function is called to actually render the frame. This can be called at interrupt time, so it is not safe to move or allocate memory in this function. You should also take care not to call functions that would do so. Your `EffectRenderFrame` function should actually render a single frame of your effect.

The `EffectBegin` function

The main tasks that the `EffectBegin` function should perform are:

- Ensure that the effect component has valid references to the current sources. If the component does not have a reference to the sources, or the sources have changed since the last call to `EffectBegin`, they must be updated.
- Ensure that the component has a valid reference to the current destination. If the component does not have a reference to the destination, or the destination has changed since the last call to `EffectBegin`, it must be updated.
- Ensure that the component has the current parameter values. If the source or destination has changed, or the component does not currently have values for the effect parameters, these parameter values are read.
- If any of the parameter values are tweened, tweening is performed to determine the actual value for those parameters.

Checking Source and Destination References

The following code checks to see if the destination has changed since the last call to the `EffectBegin` function:

```
if (p->conditionFlags & (codecConditionNewClut+
                        codecConditionFirstFrame+codecConditionNewDepth+
                        codecConditionNewDestination+codecConditionNewTransform))
```

If this evaluates to `true`, the destination has changed. This expression checks a series of flags that are passed to the `EffectBegin` function in the `conditionFlags` field of the `decompressParams` parameter. When the destination is changed, `QuickTime` sets these flags to alert the effect component to update its internal state.

The most important information that you need to store about the new destination is its base address and its `rowBytes` value. These values allow you to draw onto the destination surface.

Listing 4-2 shows an example function that stores information in the effect component's global data structure about the destination `Pixmap` passed to the function.

Listing 4-2 Storing information about a new destination frame

```
static long BlitterSetDest(BlitGlobals*glob, // input: our globals
    Pixmap *dstPixmap, // input: pixels we will draw into
    Rect *dstRect) // input: area of pixels we will draw into
{
    OSErr result = noErr;
    long offsetH,offsetV;
    char *baseAddr;
    // Calculate the based address according to the format of the
    // destination Pixmap
    offsetH = (dstRect->left - dstPixmap->bounds.left);
    if (dstPixmap->pixelSize == 16)
    {
        offsetH <<= 1; // 1 pixel = 2 bytes
    }
    else
    {
        if (dstPixmap->pixelSize == 32)
        {
            offsetH <<= 2; // 1 pixel = 4 bytes
        }
        else
        {
            result = -1; // this is a data format we can't handle
        }
    }
    offsetV = (dstRect->top - dstPixmap->bounds.top)
        * dstPixmap->rowBytes;
    baseAddr = dstPixmap->baseAddr + offsetH + offsetV;
    glob->dstBaseAddr = baseAddr;
    glob->dstRowBytes = dstPixmap->rowBytes;
    return result;
} // BlitterSetDest
```

The process for checking for new sources is broadly similar. The `CodecDecompressParams` data structure passed into the `EffectBegin` function has a field called `majorSourceChangeSeed`. This contains a seed number generated from the characteristics of the set of sources for the effect. If the sources change, the `majorSourceChangeSeed` value will also change, so the effect can store the current value in its global data structure and compare it to the current value. If they are different, the effect knows its sources have changed.

When the effect detects that one or more of its sources have changed, it must iterate through all its sources and reload information about them.

Listing 4-3 shows example code that performs these operations. Listing 4-4 shows the `BlitterSetSource` function that is called by this example code. The `BlitterSetSource` function is analogous to the `BlitterSetDest` function shown in Listing 4-2.

Listing 4-3 Checking for source changes

```
// Check to see if one or more sources have changed
```

```

if (p->majorSourceChangeSeed != glob->majorSourceChangeSeed)
{
    // grab start of input chain for this effect
    source = effect->source;
    // we can play with up to kMaxSources sources, so go get them
    while (source != nil && numSources < kMaxSources)
    {
        // now give that source to our blitter
        err = BlitterSetSource(glob, numSources, source);
        if (err != noErr)
            goto bail;
        source = source->next;
        ++numSources;
    }
}

```

Listing 4-4 Storing information about a new source frame

```

static long BlitterSetSource(BlitGlobals*glob, // input: our globals
    long sourceNumber, // input: source index to set
    CDSequenceDataSourcePtr source) // input: source value
{
    OSErr err = noErr;
    if (sourceNumber >= kMaxSources)
    {
        // too many sources for us to handle
        return noErr;
    }
    else
    {
        // a source we can handle, save it away
        err = RequestImageFormat(source, glob->width, glob->height,
            glob->dstPixelFormat);

        if (err == noErr)
        {
            glob->sources[sourceNumber].src = source;
        }
        else
        {
            glob->sources[sourceNumber].src = nil;
        }
    }
    return (err);
} // BlitterSetSource

```

Reading Parameter Values

Listing 4-5 shows how to read the value of a non-tweened parameter. The `QTFindChildByID` function is used to retrieve the atom containing the parameter value. The parameter value is then copied from the atom using the function `QTCopyAtomDataToPtr`. If the value is successfully copied, it is endian-flipped to ensure it is in native-endian format (parameter values are always stored in big-endian format). If the copy failed, a default value is provided.

The value retrieved from the parameter is stored in the component's global data structure (called, in this example, `global->blitter`). This allows the value to be used by other functions, notably the component's `EffectRenderFrame` function.

Listing 4-5 Reading a parameter value

```
{
    Ptr          data = p->data;
    QTAtom      atom;
    QTAtomID    atomID = 1;
    long        actSize;
    // Find the 'sden' atom
    atom = QTFindChildByID((QTAtomContainer) &data,
                           kParentAtomIsContainer,
                           OSTypeConst('sden'), // The name of the parameter
                           atomID,             // The ID of the parameter
                           nil);
    // Copy the parameter value from the atom
    if (QTCopyAtomDataToPtr((QTAtomContainer) &data,
                            atom,
                            false,
                            sizeof(long),
                            &((glob->blitter).scratchDensity),
                            &actSize)!=noErr)
    {
        // If the copy failed, use a default value for this parameter
        ((glob->blitter).scratchDensity) = 1;
    }
    else
    {
        // Otherwise, the copy succeeded, so endian flip and store the
        // parameter value
        ((glob->blitter).scratchDensity) =
        EndianS32_BtoN(((glob->blitter).scratchDensity));
    }
}
```

If the parameter value can contain a tweened value, you can use code similar to that shown in Listing 4-6 to retrieve the parameter value. The functions `InitializeTweenGlobals` and `CreateTweenRecord` are utility functions that Apple provides as part of the dimmer effect sample framework (see [The Sample Effect Component](#) (page 71)).

Listing 4-6 Reading a tweened parameter value

```
{
    Ptr          data = p->data;
    OSErr       err;
    long        index = 1;
    err = InitializeTweenGlobals(&glob->tweenGlobals, p);
    if (err!=noErr)
        goto bail;
    // Make our tweener, return if we already have it
    err = CreateTweenRecord(&glob->tweenGlobals,
                           &glob->percentage,
                           OSTypeConst('pcnt'), // The name of the parameter
                           1, // The ID of the parameter
                           sizeof(Fixed),

```

```

        kTweenTypeFixed,
        (void*) 0,
        (void*) fixed1,
        effect->frameTime.virtualDuration);
    if (err!=noErr)
        goto bail;
    glob->initialized = true;
}

```

Tweening Parameter Values

If you have specified that one or more of your parameter's values can be tweened, you need to implement code to perform the tweening in the `EffectBegin` function.

Listing 4-7 shows an example of tweening a parameter value. The current frame time is retrieved and subtracted from the effect's `virtualStartTime`. This calculates how far through the execution of the current effect sequence we are, expressed as a percentage.

With this information, the code then calls `QTDoTween` to interpolate the parameter value, leaving the resulting value in `glob->comp1Tween.tweenData`.

Listing 4-7 Tweening parameter values

```

wide    percentage;
// Find out how far through the effect we are
percentage = effect->frameTime.value;
CompSub(&effect->frameTime.virtualStartTime, &percentage);
// Tween our parameters and get the current value for this frame, prepare
// to render it when the EffectRenderFrame happens
{
    Fixed    thePercentage;
    if (glob->percentage.tween)
        QTDoTween(glob->percentage.tween, percentage.lo,
                  glob->percentage.tweenData, nil, nil, nil);
    thePercentage = **(Fixed**) (glob->percentage.tweenData);
    // If we are before the half-way point of this transition, we should
    // be fading the first source to black
    if (thePercentage < fixed1/2)
    {
        (glob->blitter).direction = 1;
        (glob->blitter).dimValue = FixedToInt(FixMul(IntToFixed(512),
thePercentage));
    }
    // Otherwise, we are fading up onto the new source
    else
    {
        (glob->blitter).direction = 0;
        (glob->blitter).dimValue = FixedToInt(FixMul(IntToFixed(512),
thePercentage)) - 255;
    }
}
}

```

The EffectRenderFrame Function

The `EffectRenderFrame` function is called to actually render a single frame of your effect. This is where you transform the sources of your effect into the destination frame, using the algorithm that implements your effect.

This is also where you have to handle multiple bit depth and pixel format combinations.

Internally, QuickTime stores bitmaps in a wide variety of formats. The system can handle images in a number of bit depths and with many different pixel formats. Effect components must have some ability to handle source and destination frames that are at any of the bit depths and in any of the pixel formats that QuickTime supports.

Obviously, providing a separate implementation of your effect algorithm for every combination of bit depth and pixel format could be an enormous task. Fortunately, QuickTime provides mechanisms for you to limit the number of formats you have to explicitly support.

When your effect component's `EffectSetup` function is called, it is passed information about the bit depth and pixel formats that the source frames are in. Your component should examine the formats and react in one of two ways:

- If the format is one of those which your effect does support, the `EffectSetup` function does nothing.
- If the format is not supported by your effect, `EffectSetup` returns the nearest format that is supported.

In the second case, where you do not directly support the format, QuickTime automatically creates buffers in the format returned by `EffectSetup`. The source frames are written into the buffer before `EffectRenderFrame` is called, so that source data is always available in a supported format. The destination frame is also buffered, and QuickTime automatically transforms the image into the required format for you.

This way, you only need to support a limited number of image formats, and QuickTime will ensure that `EffectRenderFrame` isn't called with data in any other format.

Note: If your effect does not handle the bit depth and pixel format combination passed to the `Setup` function, and it requests an alternative format, QuickTime generates new offscreen buffers for each source and destination frame your effect uses. This will result in a memory and execution time overhead for your effect. If you want your effect to execute quickly in a wide range of circumstances, your effect should explicitly handle as many bit depth and pixel format combinations as possible.

Handling Multiple Formats

Although you can write separate versions of your effect algorithm for each combination of bit depth and pixel format, Apple recommends that you implement your effect algorithm once for each bit depth. You should then use the Apple-supplied **blit macros** to automatically generate versions of these implementations for each supported pixel format. This significantly reduces the number of separate implementations you have to maintain, and allows easy support of multiple pixel formats.

The blit macros are contained in the file `BlitMacros.h`, which is included with the sample effect framework code.

To use the blit macros in your effect component, you must store each bit depth implementation of your effect algorithm in a separate file. These files are then included into the effect component's main source code file multiple times, once per pixel format supported. Each inclusion is surrounded by `#define` statements that define the pixel format version to be generated.

Each file uses a `#include` statement to include `BlitMacros.h`, and all operations that read pixels from a source buffer or write pixels to the destination buffer are performed using appropriate macros.

The macros are automatically converted to the correct operations for the pixel format when the file is included into the main source code. This generates a version of the algorithm for each pixel format.

Finally, code is put into place in the effect component's `EffectRenderFrame` function, which calls the appropriate generated algorithm according to the current bit depth and pixel format of the source buffers.

Implementing a Bit-depth Specific Version of Your Algorithm

Listing 4-8 shows an example implementation of an effect algorithm. The code uses the blit macros to read pixels from the source frame and write them to the destination frame. This example shows a filter that changes a single source. It also shows how to read and alter a single pixel at a time; other effects may handle multiple pixels at a time for efficiency.

The sample shows the following operations for each pixel of the source frame:

- Retrieving the next pixel from the source, using the `Get16` (which reads a 16-bit pixel from a memory address) and `cnv16SPFto16RG` (which converts a 16-bit pixel in the current pixel format to the standardized 16-bit ARGB format) macros to handle pixel format conversion;
- Decomposing the pixel into alpha, red, green and blue components;
- Reassembling the alpha, red, green and blue components into a standardized ARGB pixel value;
- Writing the pixel value to the destination buffer, using the `cnv16RGto16DPF` (which converts the 16-bit standardized format pixel back into the current buffer's 16-bit pixel format) and `Set16` (which writes a 16-bit pixel to a memory address) macros to handle pixel format conversion.

The actual effect implementation, which would alter the alpha, red, green and blue values of each pixel according to the effect specification, is not shown in this example code.

Listing 4-8 A sample effect algorithm for 16-bit frames

```
#include <BlitMacros.h>
void EffectFilter16(BlitGlobals *glob);
void EffectFilter16(BlitGlobals *glob)
{
    long    height = glob->height;        // Local copy of the height of
                                         // the buffers
    UInt16 *srcA = glob->sources[0].srcBaseAddr; // Local pointer to
                                                // the first source image
    UInt16 *dst = glob->dstBaseAddr; // Local pointer to the
                                     // destination

    long    srcABump;
    long    dstBump;

    // Work out the source and destination "bumps". The rowBytes value
    // gives you the number of bytes in each scanline of an image. This
```

```

// is not necessarily the same as the number of pixels in a scanline
// multiplied by the number of bytes each pixel occupies. When
// we copy pixels from source to destination, via our effect
// algorithm, we need to account for this discrepancy. The following
// lines pre-calculate the differences.
srcABump = glob->srcRowBytes - (glob->width * 2);
dstBump = glob->dstRowBytes - (glob->width * 2);
// Now, for every scanline in the source image we are dealing with...
while (height--)
{
    long    width = glob->width;
    // ...iterate through every pixel in that scanline
    while (width--)
    {
        UInt16    thePixelValue;
        // Retrieve the next pixel value
        thePixelValue = Get16(srcA);
        srcA++;
        // Call to blit macros to ensure the pixel format is
        // converted appropriately
        cnv16SPFto16RG(thePixelValue);
        // Get the alpha, red, green and blue values of the pixel
        alpha = 0x8000 & thePixelValue;
        red   = (thePixelValue & 0x7C00) >> 10;
        green = (thePixelValue & 0x03E0) >> 5;
        blue  = (thePixelValue & 0x001F) >> 0;
        // IMPLEMENT YOUR EFFECT ALGORITHM HERE ON EACH PIXEL
        // Re-assemble the A, R, G and B values into a 16-bit
        // destination pixel
        thePixelValue = alpha | (red << 10) | (green << 5)
                          | (blue << 0);
        // Set the destination pixel, first passing it through the
        // appropriate blit macro to
        // ensure the correct pixel format conversion is performed
        cnv16RGto16DPF(thePixelValue);
        Set16(dst, thePixelValue);
        dst++;
    }
    // Bump the source and destination pointers we are using, to
    // avoid problems when moving from one scanline to the next
    srcA = (void *) (((Ptr) srcA) + srcABump);
    dst = (void *) (((Ptr) dst) + dstBump);
}
}

```

Including the Bit-depth Implementations in Your Effect Code

Once you have produced separate implementations of your effect algorithm for each bit depth you support, you need to include these in your main effect source code file. Each bit depth implementation is included once for every pixel format you support.

Listing 4-9 shows statements to include the 16-bit implementation of the effect into the main effect source code file. The implementation is included three times, for the following pixel formats:

- Big-endian 555 RGB

- Little-endian 555 RGB
- Little-endian 565 RGB

The result of the code in Listing 4-9 is that your effect source code contains three separate versions of the effect algorithm for handling 16-bit sources. These are named `EffectFilter16BE555`, `EffectFilter16LE555`, and `EffectFilter16LE565`, respectively.

Listing 4-9 Including the 16-bit implementation into the main effect source code

```
// 16-bit, Big Endian 555 pixel format
#define EffectFilter16 EffectFilter16BE555
#define srcIs16BE555 1
#define dstIs16BE555 1
#include "EffectFilter16.c"
#undef EffectFilter16
#undef srcIs16BE555
#undef dstIs16BE555
// 16-bit, Little Endian, 555 pixel format
#define EffectFilter16 EffectFilter16LE555
#define srcIs16LE555 1
#define dstIs16LE555 1
#include "EffectFilter16.c"
#undef EffectFilter16
#undef srcIs16LE555
#undef dstIs16LE555
// 16-bit, Little Endian, 565 pixel format
#define EffectFilter16 EffectFilter16LE565
#define srcIs16LE565 1
#define dstIs16LE565 1
#include "EffectFilter16.c"
#undef EffectFilter16
#undef srcIs16LE565
#undef dstIs16LE565
```

Calling the Effect Implementations from `EffectRenderFrame`

Finally, you must provide code inside your `EffectRenderFrame` function to call the appropriate implementation of your effect algorithm, depending on the pixel format and bit depth of the source frames you are dealing with. Listing 4-10 shows how to do this for the 16-bit pixel formats.

Listing 4-10 Calling pixel format specific versions of the 16-bit effect implementation

```
switch (glob->dstPixelFormat)
{
    case k16BE555PixelFormat:
        EffectFilter16BE555(glob);
        break;
    case k16LE565PixelFormat:
        EffectFilter16LE565(glob);
        break;
    case k16LE555PixelFormat:
        EffectFilter16LE555(glob);
        break;
}
```

The code to handle the 32-bit pixel formats is an easy extension of the code shown in this section, and can be found in the sample effect component included in the QuickTime SDK and described in detail in the next section.

The Sample Effect Component

This section introduces you to the sample effect component supplied as part of the QuickTime SDK. It takes you through the parts of the code that you will need to change in order to implement your own effect component.

The Dimmer Effect

The sample effect described in this section is a dimmer effect. This simple effect fades the first source to black, then fades up to show the second source. The source code for this effect is provided as a CodeWarrior Pro project as part of the QuickTime SDK.

The Standard Effect Framework

Much of the code required to implement an effect component is the same for all components. Apple has provided a framework of code that you can adapt to create your own effect components. In most cases, you only have to change limited portions of the framework code to create your new component.

Structure of the Framework

The effect framework is one approach to writing effects components. It has been designed to provide most of the basic code required to implement an effect component, leaving the implementation of the effect algorithm to you. It should be possible to write most effects using the framework, though you may need to adapt the framework code for more complex effect components.

The framework implements the required component functions for an effect. The main body of the framework is the `Effect.c` file, which contains the source code for the framework and an implementation of the dimmer effect. This file is made up of four main parts:

- The **global data structures** used by the framework. You will need to update some of the data structures to reflect the capabilities of the effect you are implementing.
- The **dispatcher** is the entry point to your component. Because all effects components have the same set of component functions, you should not need to alter the dispatcher.
- The **internal functions** are the set of functions that actually execute your effect. This is where most of your own code will be added.
- The **component functions** are the standard functions called by the dispatcher. These functions call the internal functions to actually execute the effect. For most effects, you won't need to change much code in this section.

Naming Conventions

All the function and data structure names in the framework are arbitrary. The names have been chosen to reflect their purpose, but you are free to change the names, as long as they remain internally consistent.

If you choose to change the names of the component functions, you will have to change the `CALLCOMPONENT_BASENAME` `#define` in `Effect.c`. This defines the root of the name used for component functions. For example, if `CALLCOMPONENT_BASENAME` is set to `SlideEffect`, then the Open component function must be called `SlideEffectOpen`, the Close component function must be called `SlideEffectClose`, and so forth.

Apple recommends that you do not change the names used in the framework.

Writing an Effect Component Using the Framework

The effect component framework is provided for you to simplify the development of QuickTime video effects components.

The QuickTime SDK includes the folder `DimmerEffect`, which contains the framework, complete with associated resources and makefiles. See the `ReadMe` file in the `DimmerEffect` folder for full installation and use instructions.

To adapt the dimmer framework to create your own component, search through the source code file `Effect.c` for the comments `CHANGE`. These comments mark the sections of the source code you will need to change to write your own effect.

The following sections take you through the specific changes you need to make to the framework. All the changes except the last, which implements the actual effect algorithm, are made to the `Effect.c` file.

Synchronous vs. Asynchronous Processing

The first change you may need to make is to the following `#define`:

```
#define kMaxAsyncFrames 0
```

This value defines the number of frames that can be queued for asynchronous rendering by this effect. If your effect declares that it can handle more than 0 asynchronous frames, frames may be queued for rendering. If you wish to render synchronously, set `kMaxAsyncFrames` to 0; otherwise set it to the number of frames that can be held in the queue.

Defining the Number of Sources

Most effects require one or more sources to operate on, though some effects (such as Apple's fire effect) operate without any sources. The dimmer effect uses two sources: the first is the source to fade to black, the second is the source to fade up on. Most effects transition between two sources. Sometimes, effects control the transition between two scenes by blending in one or more other sources, in which case the effect may require three or more sources. You may also want to implement a filter effect that has only a single source and produces a transformed version of that source.

You set the value of `kMaxSources` to the maximum number of sources required by the effect. Effects that can take more than one source should be prepared to handle the case when fewer than the maximum number of sources are actually provided. For example, if your effect expects two sources to transition between and a third source to use as a mask, your code must handle the case where only the two transition sources are provided. In this case you should use a default mask instead of a third source.

Adding to the Global Data Structures

The framework defines two global data structures: `BlitGlobals` and `EffectGlobals`. The `BlitGlobals` structure holds information related to drawing a single frame of the effect, while the `EffectGlobals` holds data for the entire effect as it is executed. These data structures are global to an instance of the effect component. That is, if you have multiple instances of the component opened, each instance gets its own copy of both data structures.

You can add fields to the `BlitGlobals` data structure to hold information specific to your effect. A set of standard fields are already defined, which hold information used by the framework. You can add your own effect-specific fields between the `CHANGE` and `END CHANGE` comments.

The example defines two fields, `dimValue` and `direction`. These hold the current dim value for the effect and a flag indicating whether it is fading down or up, respectively. Because the dimmer fades the first scene down to black then fades up on the second scene, it also needs to store the dim value between individual frames. This value is stored in the `dimValue` field of `BlitGlobals`.

You can also add fields to the `EffectGlobals` structure. Generally, you will read the values for the parameters to your effect in these fields so that they can be referenced while the effect executes.

Preflighting the Blitter

The internal function `BlitterPreflight` is called from `EffectSetup` before the first frame of the effect is rendered. This function's main task is to validate the bit depth that the effect is being requested to support.

The bit depth that the effect is being asked to operate at is passed in the `depth` parameter to `BlitterPreflight`. The function should return in the same parameter the bit depth at which it wants to operate.

For example, the dimmer effect can operate on 16-bit or 32-bit sources. If either of these values is passed in, it simply returns `depth` unaltered. If any other bit depth is requested, it sets `depth` to 16, the default bit depth for this effect.

Your effect should validate the bit depth passed in a similar way. Apple recommends that your effect support at least 16- and 32-bit depths.

When you set the `depth` parameter to a different value than it was on entry to `BlitterPreflight`, QuickTime creates an offscreen buffer for the sources and destination of the effect. All data is passed through these offscreen buffers, to ensure that your effect only sees data in a format it can handle.

Setting the Destination

The `BlitterSetDest` function is called from `EffectBegin` and is passed the effect's destination, in the form of a `PixelFormat`. The `BlitterSetDest` function should calculate the base address and `rowBytes` values for the destination and store these in the `BlitGlobals` data structure for future reference.

You need to make changes to this function only if your effect supports destinations in bit depths other than 16-bit and 32-bit.

The `BlitterRenderFrame` function

This function calls the functions that implement your effect algorithm. The function names to be called are those generated by the blit macros.

The example code supports the three most common pixel formats in 16-bit and 32-bit. If your effect needs to support other bit depths or pixel formats, you need to update the `switch` statement in this function so that the appropriate drawing functions are called.

The `EffectsFrameClose` function

This function is called when the client software has finished using your component. At this time, your component should dispose of any memory it allocated. In particular, you should call `DisposeTweenRecord` for each tween record you allocated and then call `DisposeTweenGlobals`.

Reading the Effect Parameters

The parameters of the effect are read in the `EffectsFrameEffectBegin` function. Your effect should read its parameter values in the section between the `CHANGE` and `END CHANGE` comments, reading either non-tweened or (more frequently) tweened values. Example code for both these cases is given in [Reading Parameter Values](#) (page 64).

Once you have read in the parameter values, you need to tween those parameters that contain tween records. This code should be placed between the second pair of `CHANGE` and `END CHANGE` comments. Again, example code to do this is supplied, see [Tweening Parameter Values](#) (page 66).

Implementing your Effect

The last stage in adapting the framework is to implement your effect algorithm. You need to provide one implementation per bit depth that your effect explicitly supports, and each implementation must be placed in a separate file. These files are named `EffectFilter16.c`, `EffectFilter32.c`, and so forth.

The dimmer effect code provides an example of the pixel manipulations that an effect will typically perform, and shows how to use the blit macros to support multiple pixel formats at a given bit depth.

Clearly, the details of these routines are entirely dependent on the effect being implemented.

Adding an 'atms' Resource to your Component

The 'atms' resource for your effect contains two sets of information. The first set contains the effect information that is used to construct the standard parameters dialog box. This includes items such as the name of your effect and optional copyright information.

The second set contains the parameter information, which is a description of each parameter that your effect takes. If your effect does not take parameters, there is no information in this set.

The structure of an 'atms' resource is as follows:

```
resource 'atms' (kEffectatmsRes) {
    7,
    {
        // The resource body goes here
    };
};
```

The header for this resource contains two items: the resource ID, and the number of root level atoms the resource contains.

The first line contains the ID of the 'atms' resource. In this example, the identifier that is used (`kEffectatmsRes`) is also used in the call to `GetComponentResource` in Listing 4-1. This ensures that the right 'atms' resource is read by QuickTime.

The second line contains the number of root atoms in the resource. Each 'atms' resource contains a number of atoms. The number in the second line must contain a count of the number of first-level atoms in the resource.



Warning: It is critical that you update this number if you add or delete atoms from your 'atms' resource. If this number is larger than the number of atoms in your effect, your effect component can cause QuickTime to crash. If this number is smaller than the number of atoms actually in the resource, users will not be able to adjust the values of some parameters.

The body of the 'atms' resource consists of a number of atom declarations. Each declaration has a header that contains

- the atom name
- the atom ID
- the number of children in the declaration

Each header is followed by the atom's data, which is one or more typed values, such as a `string` or a `long`, or a set of child atoms.

Listing 4-11 shows an example atom that contains a single typed value as its data. Note that the value is a type followed by the data itself. The number of children of the atom is declared as `noChildren` because the atom contains a typed value.

Listing 4-11 An example 'atms' atom declaration

```
kParameterTitleName, kParameterTitleID, noChildren,
```

```
{
    string { "Dimmer2 Effect Parameters" };
};
```

The Standard Information in an 'atms' Resource

The standard information stored in an 'atms' resource is made up of three required atoms and five optional atoms.

The three required atoms are

Term	Definition
kParameterTitleName	a string used as the title of the standard parameters dialog box. An example of a kParameterTitleName atom declaration is shown in Listing 4-11.
kParameterWhatName	an OSType containing the name of this effect component.
kParameterSourceCountName	a long integer containing the maximum number of sources that the effect can take.

The five optional atoms are

Term	Definition
kParameterAlternateCodecName	an OSType containing the unique identifier of another effect component that should be used to replace this effect if this effect cannot be used.
kParameterInfoLongName	a string containing the long version of the name of the effect.
kParameterInfoCopyright	a string containing a copyright statement for the effect.
kParameterInfoDescription	a string containing a brief description of what the effect does.
kParameterInfoWindowTitle	a string containing the title of the window that displays the information contained in the optional atoms.

The Parameter Information in an 'atms' Resource

For each parameter of the effect, your 'atms' resource must contain a set of atoms in the 'atms' resource that describes that parameter. This description includes the name of the parameter, the type and range of values it can take, and hints on appropriate user interface element for setting this parameter.

A complete description of the information you need to provide for each parameter can be found in [The Parameter Description Format](#) (page 78).

For a basic parameter, there are five atoms that you should supply:

Term	Definition
kParameterAtom- TypeAndID	contains the type and ID of the parameter (an OSType and a long integer, respectively), the atom flags for the parameter, and a string containing the name of the parameter.
kParameterDataType	a long integer containing the type of the parameter, such as kParameterTypeDataFixed.
kParameterDataRange	a set of typed values describing the range of values the parameter can take. The number and type of values you supply depend on the value of the kParameterDataType atom.
kParameterDataBehavior	two long integer values containing the behavior type, such as kParameterItemControl for a slider, and any flags, such as kAtomNotInterpolated for a parameter that cannot be tweened.
kParameterData- DefaultItem	the default value of the parameter. Again, the type of this value will depend on the type of the kParameterDataType atom. This atom must be a correctly-formatted parameter atom that can be passed back to your component by client software without modification.

An example of a basic parameter description is shown in Listing 4-12.

Listing 4-12 An example set of parameter description atoms

```
kParameterAtomTypeAndID, 101, noChildren,
{
    OSType { "sden" }; // atomType--the name of this parameter
    long { "1" }; // atomID--this is atom number 1
    kAtomNotInterpolated; // atomFlags--this parameter cannot be tweened
    string { "Scratch Density" }; // atomName--the name of the parameter
    // as it will appear in the standard parameters dialog box
};
kParameterDataType, 101, noChildren,
{
    kParameterTypeDataLong; // dataType--this parameter contains a
    // long value
};
kParameterDataRange, 101, noChildren,
{
    long { "0" }; // minimumValue
    long { "25" }; // maximumValue
    long { "1" }; // scaleFactor--no scaling is applied to this
    // parameter
    long { "0" }; // precision--0 indicates that this parameter is
    // not a floating-point value
};
kParameterDataBehavior, 101, noChildren,
{
    kParameterItemControl; // behaviorType--this parameters should be
    // represented by a slider
    long { "0" }; // behaviorFlags - no flags
};
kParameterDataDefaultItem, 101, noChildren,
```

```

{
    long { "5" };           // the default value of the parameter
};

```

The Parameter Description Format

The parameter description data structure is a `QTAtomContainer` structure that, when filled out by the `ImageCodecGetParameterList` call, contains a set of `QTAtoms` for each parameter of the effect. These atoms define the base type of the parameter, the legal range of values that can be stored in it, and hints for displaying a user interface to set values for the parameter.

The atoms in a parameter description are described in the following sections. The order in which the atoms are stored in the `QTAtomContainer` structure is important. Applications should present parameters to the user in the same order that they are contained in the parameter description.

Each of the atom types in a parameter description has a name; you will find constants for these in `ImageCodec.h`. You should use these constants when retrieving atoms from the data structure. The data stored in the atoms of the parameter description is structured, and the `struct` definitions are given in the atom descriptions below.

Many of the atoms must be present to create a valid parameter description. Some are optional, as noted.

Parameter Atom Type and ID

This atom contains information about the type and ID of the parameter. The data is contained in the following structure:

```

typedef struct
{
    QTAtomType  atomType;
    QTAtomID    atomID;
    long        atomFlags;
    Str255      atomName;
} ParameterAtomTypeAndID;

```

Term	Definition
<code>atomType</code>	This field contains either a unique identifier for the parameter or the value <code>kNoAtom</code> . The unique identifier is a four character <code>OSType</code> that you use to retrieve the parameter's value. If this field contains <code>kNoAtom</code> , the "parameter" being described is actually a group description; groups are described in "Special Description Types" [link s Creating New Video Effects].
<code>atomID</code>	This field contains the ID of this parameter.

Term	Definition
atomFlags	This field can contain one of the predefined values: <code>kAtomNoFlags</code> , <code>kAtomNotInterpolated</code> , or <code>kAtomInterpolateIsOptional</code> . If it contains <code>kAtomNotInterpolated</code> , the user interface allows users to enter only a single value for this parameter, and this value remains constant while the effect is playing. If it contains <code>kAtomNoFlags</code> , the user interface allows users to enter a set of values for the parameter. This set of values are stored in a tween atom, and the value of the parameter is interpolated between these values during effect playback. If it contains <code>kAtomInterpolateIsOptional</code> , the user interface defaults to allowing a single value for the parameter. If the user interface supports an "advanced" mode of operation, then a tween value can be entered for this parameter when the user interface is in this mode. An example of an advanced mode is the standard parameters dialog box: if you hold down the option key while selecting an effect, any parameters that have the <code>kAtomInterpolateIsOptional</code> flag set will allow a tween value to be entered.
atomName	The name of this parameter. This string value is used as the name of the control displayed in the standard parameter dialog box to enter a value for this parameter. This atom is required.

Special Description Types

If the parameter atom type and ID atom of a parameter description contains the constant `kNoAtom`, this indicates that the value being described is not a parameter to the effect but is a group. Besides groups, two further special cases are covered in the following sections: enumeration lists and source counts.

Groups

It is sometimes useful to treat a set of parameters as a group. For example, you might want to label a group of parameters that jointly control something, align a group of controls, or enclose a set of parameters in a box. The grouping mechanism allows you to specify a set of parameters and the attributes that are applied to the group.

If the parameter data type and ID atom of a description contains child atoms, rather than data, it defines a group. A group is a set of related atoms, where the relationship amongst them can be based on attributes such as:

- layout; for example, the group is a set of text labels that should be aligned.
- spatial; for example, the items in the group should be placed side by side to optimize dialog box layout.
- naming; the items in the group are related controls that should be displayed under a single heading in the dialog box.
- usage; a pair of long integers may together specify a coordinate. In this case, they can be grouped together and the group's parameter data usage atom set to `kParameterUsagePoint`.

Groups can be nested within one another as needed. Groups can optionally have a name, which allows your application to place grouped parameters within a panel or tabbed group under that name.

Listing 4-13 shows an example of a group, which in this case contains a single parameter description.

Listing 4-13 An example group atom from an 'atms' resource definition.

```

kParameterAtomTypeAndID, 100, noChildren,
{
    OSType { "none" }; // Use 'none' as this is not a real parameter
    long { "0" };
    kAtomNoFlags;
    string { "" };
};
kParameterDataBehavior, 100, noChildren,
{
    kParameterItemGroupDivider; // Use a divider to separate this group
    kGroupNoFlags;
};
kParameterDataType, 100, 1*5, // 1 parameter * 5 atoms to describe each
                               //parameter
{
};
kParameterAtomTypeAndID, 3, noChildren,
{
    OSType { "pMul" };
    long { "1" };
    kAtomNotInterpolated;
    string { "Pre-multiply color" };
};
kParameterDataType, 3, noChildren,
{
    kParameterTypeDataRGBValue;
};
kParameterDataRange, 3, noChildren,
{
    short { "0" };
    short { "0" };
    short { "0" };
    short { "65535" };
    short { "65535" };
    short { "65535" };
};
kParameterDataBehavior, 3, noChildren,
{
    kParameterItemColorPicker;
    long { "0" };
};
kParameterDataDefaultItem, 3, noChildren,
{
    short { "65535" };
    short { "65535" };
    short { "65535" };
};
};

```

Enumeration Lists

When an enumerated type is required for a parameter value, a new enumeration list is placed directly into the root atom container. Enumeration lists are arrays of name-and-value pairings in the following format:

```
typedef struct
```



```

{
    long    value;
    Str255 name;
} EnumValuePair;

typedef struct
{
    long          enumCount; // number of enumeration items to follow
    EnumValuePair values[1]; // values and names for them
} EnumListRecord;

```

The type of an enumeration list atom is `kParameterEnumList('enum')`. Listing 4-14 shows an enumeration list that contains three elements.

Listing 4-14 An example enumeration list from an 'atms' resource definition

```

kParameterEnumList, 1, noChildren,
{
    long { "3" }; // No of elements in the enum
    long {"1"}; string { "Straight Alpha" };
    long {"2"}; string { "Pre-multiply Alpha" };
    long {"3"}; string { "Reverse Alpha" };
};

```

Source Count

The source count atom (`kParameterSourceCountName, 'srCs'`) contains a single long integer value that defines the maximum number of sources that this effect can accept. The atom is always placed in the root atom container of the parameter description.

The source count atom is required.

Parameter Data Type

This atom defines the type of the data for this parameter. It contains data in the following structure:

```

typedef struct
{
    OSType dataType;
}

```

Term	Definition
dataType	This field contains the type of the value that is stored in this parameter. This can be one of the following values:

Term	Definition
kParameterTypeDataText	editable text item
kParameterTypeDataLong	integer value

Term	Definition
<code>kParameterTypeDataEnum</code>	enumerated lookup value
<code>kParameterTypeDataFixed</code>	fixed point value
<code>kParameterTypeDataDouble</code>	IEEE 64 bit floating point value
<code>kParameterTypeDataBitField</code>	bit field (Boolean) value
<code>kParameterTypeDataRGBValue</code>	RGBColor data
<code>kParameterTypeDataImage</code>	reference to an image

This atom is required.

Parameter Alternate Data Type

This atom defines a preferred data type for the parameter. If the system your application is running on does not support this preferred data type, the data type specified in the parameter data type atom will be used instead.

Use the alternate data type atom if you would prefer to use a data type that is not supported on all platforms, and use the parameter data type atom to specify a fall-back data type for systems that do not support your preferred data type.

For example, if the parameter alternate data type is `kParameterTypeDataColorValue`, the parameter holds a value of type `CMColor` on systems that have the `ColorSync` extension. On systems that do not have `ColorSync`, whatever is specified in the parameter data type (such as an `RGBValue`) is used instead.

This atom's data is stored in a `ParameterAlternateDataType` data structure, which in turn relies on the `ParameterAlternateDataEntry` data structure.

```
typedef struct
{
    OSType      dataType;          // The type of the data
    QTAtomType  alternateAtom;    // The atom to use for alternate data
} ParameterAlternateDataEntry;
typedef struct
{
    long        alternateCount;
    ParameterAlternateDataEntry  alternates[];
} ParameterAlternateDataType;
```

Term	Definition
<code>dataType</code>	This field in the <code>ParameterAlternateDataEntry</code> structure can take one of the following values:

Term	Definition
kParameterTypeDataColorValue	CM color data
kParameterTypeDataCubic	Cubic Beziars
kParameterTypeDataNURB	Nurbs

The parameter alternate data type atom is optional.

Parameter Data Range

The Parameter Data Range atom defines the legal range of values that the parameter can take. It also defines a scaling constant that defines how the legal range of values can be translated into a range that is more suitable for display in a user interface. For example, a value with a range of 0-255 might be scaled as 0-100 for user input.

The atom's data is structured as a `RangeRecord`, defined below. The exact format of this data depends on the data type of the parameter being described.

```
// 'text'
typedef struct
{
    long    maxChars;    // Maximum length of the string
    long    maxLines;    // Number of editing lines (typically 1)
} StringRangeRecord;
// 'long'
typedef struct
{
    long    minValue;    // Minimum value the long can be
    long    maxValue;    // Maximum value the long can be
    long    scaleValue;   // Scaling constant
    long    precisionDigits; // number of digits of precision
                                // when editing via typing
} LongRangeRecord;
// 'enum'
typedef struct
{
    long    enumID;      // The ID of the 'enum' atom in the
                        // root container to search
} EnumRangeRecord;
// 'fixd'
typedef struct
{
    Fixed    minValue;    // Minimum value the Fixed can be
    Fixed    maxValue;    // Maximum value the Fixed can be
    Fixed    scaleValue;   // Scaling constant
    long    precisionDigits; // number of digits of precision
                                // when editing via typing
} FixedRangeRecord;
// 'doub'
typedef struct
{
    QTFloatDouble    minValue;    // Minimum value of parameter
```

```

    QTFloatDouble   maxValue;           // Maximum value of parameter
    QTFloatDouble   scaleValue;        // Scaling constant
    long            precisionDigits;    // number of digits of precision
                                           // when editing via typing
} DoubleRangeRecord;
// 'bool'
typedef struct
{
    long            maskValue; // value to mask on/off to set/clear the
                               // boolean
} BooleanRangeRecord;

// 'rgb '
typedef struct
{
    RGBColor       minColor;           // Minimum value the RGBColor can be
    RGBColor       maxColor;           // Maximum value the RGBColor can be
} RGBRangeRecord;
// The RangeRecord data structure is the union of all of the above
typedef struct
{
    union
    {
        LongRangeRecord   longRange;
        EnumRangeRecord   enumRange;
        FixedRangeRecord  fixedRange;
        DoubleRangeRecord  doubleRange;
        StringRangeRecord  stringRange;
        BooleanRangeRecord  booleanRange;
        RGBRangeRecord     rgbRange;
    } u;
} RangeRecord;

```

The `minValue` and `maxValue` fields of the `DoubleRangeRecord` data structure can take, in addition to an actual `QTFloatDouble` value, the following predefined values:

- `kNoMinimumDouble`; ignore the minimum value
- `kNoMaximumDouble`; ignore the maximum value
- `kNoScaleDouble`; don't perform any scaling of value

The `minValue` and `MaxValue` fields of the `LongRangeRecord` data structure can take, in addition to an actual long integer value, the following predefined values:

- `kNoMinimumLongFixed`; ignore minimum value
- `kNoMaximumLongFixed`; ignore maximum value
- `kNoScaleLongFixed`; don't perform any scaling of value
- `kNoPrecision`; allow as many digits as format

The `Parameter Data Range` atom is required, except for group descriptions.

Parameter Data Behavior

The Parameter Data Behavior atom contains user interface hints that suggest to the client application how a parameter should be displayed.

Note: These user interface hints can be ignored by your application if you have a specific interface style you wish to implement. However, Apple recommends that you use the editing mechanisms suggested for the parameter whenever possible. If your application does not use the suggested behavior, you will present an inconsistent and potentially confusing interface to your users.

```
typedef struct
{
    QTAtomID    groupID;
    long        controlValue;
} ControlBehaviors;

typedef struct
{
    OSType      behaviorType;
    long        behaviorFlags;
    union
    {
        ControlBehaviorscontrols;
    } u;
} ParameterDataBehavior;
```

Term	Definition
behaviorType	This field contains a value that specifies a user interface for editing the parameter's value. This field should contain one of the following pre-defined values:

Term	Definition
kParameterItemEditText	the parameter should be edited using an edit text field.
kParameterItemEditLong	the parameter should be edited using an edit text field that only accepts numerical entries.
kParameterItemEditFixed	the parameter should be edited using an edit text field that accepts floating-point numerical entries.
kParameterItemPopUp	the parameter should be edited using a pop-up menu. This data behavior should only be used with parameters whose data type is kParameterTypeDataEnum; the pop-up menu is populated from the enumeration values.
kParameterItemRadioCluster	the parameter should be edited using a group of radio buttons. This data behavior should only be used with parameters whose data type is kParameterTypeDataEnum; the radio buttons are created from the enumeration values

Term	Definition
<code>kParameterItemCheckBox</code>	the parameter should be edited using a checkbox This data behavior should only be used with parameters whose data type is <code>kParameterTypeDataBitField</code> .
<code>kParameterItemControl</code>	the parameter should be edited using a standard control appropriate to the data type of the parameter. For parameters that accept a scalar value, such as a <code>Fixed</code> or a <code>Long</code> , the control used is a slider.
<code>kParameterItemLine</code>	a horizontal line is drawn in above the control that manipulates this parameter's value.
<code>kParameterItemRectangle</code>	a rectangle is drawn around the control that manipulates this parameter's value.
<code>kParameterItemColorPicker</code>	the parameter should be edited using a color swatch and picker.
<code>kParameterItemGroupDivider</code>	start of a new group of items.
<code>kParameterItemStaticText</code>	the parameter's name is displayed as a static text field.
<code>kParameterItemDragImage</code>	the parameter should be edited as an image that accepts drag and drop entry of new images.
<code>kParameterItemDragPath</code>	the parameter should be edited as a path display that allows the user to drag out a new path.
<code>behaviorFlags</code>	This field can take one or more of the following values:

Flag	Definition
<code>kGraphicsNoFlags</code>	no options for graphics.
<code>kGraphicsFlagsGray</code>	any lines or rectangles that are drawn have a grayscale appearance. If this option is not set, lines and rectangles are drawn in black.
<code>kGroupNoFlags</code>	no options for the group.
<code>kGroupAlignText</code>	the controls in the group are aligned.
<code>kGroupSurroundBox</code>	the controls in the group are surrounded with a box.
<code>kGroupMatrix</code>	display the controls in the group in a matrix, if such an arrangement is possible.
<code>kGroupNoName</code>	do not display the name of the group.

The `behaviorFlags` values allow you to optionally show or hide a group depending on the value entered into a parameter. This allows you to express simple conditionals within a standard parameters dialog box. For example, you may want a pop-up menu with a set of fixed options, and an 'Others...' option; if the user chooses 'others', a text edit field is enabled to allow users to enter their own value.

To do this, you can use the `kDisableWhenLessThan` flag to specify that the group containing the text control is disabled when the user chooses any value in the pop-up menu that is less than the last, 'Others...' option.

The following flags are available to control selective disabling of groups. For each of these flags, the ID of the group to be disabled is stored in the `groupID` field of the `controls` data structure. The value that is used in the comparison operation is stored in the `controlValue` field of the `controls` data structure.

Flag	Definition
<code>kDisableWhenNotEqual</code>	When the value chosen for this parameter is not equal to <code>controlValue</code> , disable the group <code>groupID</code> .
<code>kDisableWhenEqual</code>	When the value chosen for this parameter is equal to <code>controlValue</code> , disable the group <code>groupID</code> .
<code>kDisableWhenLessThan</code>	When the value chosen for this parameter is less than the <code>controlValue</code> , disable the group <code>groupID</code> .
<code>kDisableWhenGreaterThan</code>	When the value chosen for this parameter is greater than the <code>controlValue</code> , disable the group <code>groupID</code> .

Note: You can only disable groups, not individual parameters. However, you can create a group with no visual attributes that contains a single parameter.

The parameter data behavior atom is required.

Parameter Data Usage

The parameter data usage atom defines the intended use of the data in the parameter. This information can be used by your application to provide a more appropriate user interface for a parameter or group of parameters. For example, if your application knows that a set of four long integer values actually represent a rectangle, it can present a graphical display of the rectangle, rather than simply displaying four numeric input fields.

The data in this atom is stored in the following data structure:

```
typedef struct
{
    OSType  usageType;
} ParameterDataUsage;
```

Term	Definition
<code>usageType</code>	This field defines the actual use that a parameter or group of parameters. It can take one of the following values:

Term	Definition
kParameterUsagePixels	The parameters in the group contain a set of pixels.
kParameterUsageRectangle	The parameters in the group contain the top-left and bottom-right coordinates of a rectangle.
kParameterUsagePoint	The parameters in the group contain the coordinates of a point.
kParameterUsage3DPoint	The parameters in the group contain the X,Y,Z coordinates of a 3D point.
kParameterUsage3by3Matrix	The parameters in the group contain a 3x3 matrix of values.
kParameterUsageDegree	The parameter contains degrees.
kParameterUsageRadians	The parameter contains radians.
kParameterUsagePercent	The parameter contains a percentage.
kParameterUsageSeconds	The parameter contains seconds.
kParameterUsageMilliseconds	The parameter contains milliseconds.
kParameterUsageMicroseconds	The parameter contains microseconds.

The parameter data usage atom is optional.

Parameter Data Default Item

The parameter data default item atom contains the default value for the parameter. This value is stored in a QuickTime atom and can be copied directly into the parameter or into an effect description; an application does not need to understand the contents or format of the atom in order to do this.

The parameter data default item atom is required, except for group descriptions.

Tweening Parameters

An important property of effect parameters is that many can be tweened, and some must be tweened. Tweening is QuickTime's general purpose interpolation mechanism (see *QuickTime Media Types and Media Handlers Guide*). This allows the value of the parameter to change as the effect executes.

For example, the slide effect built into QuickTime (see [Slide](#) (page 89)) has an angle parameter. This controls the angle from which the second source will slide over the first during the execution of the effect. If this parameter contains a single value, the second source will slide over the first in a straight line from the selected angle. However, if the parameter contains two values, the angle will be interpolated between these values during the execution of the effect. This allows you to specify a curved slide effect.

In fact, any valid tween record can be specified as the parameter value, not just records containing pairs of values. The QuickTime tweening mechanism supports tween records that contain more than two values and that specify the interpolation algorithm used to produce intermediate values. However, the standard parameters dialog box allows only a pair of values to be entered, and the appropriate default interpolator is used. The standard parameter dialog box presents the user with a pair of values for parameters that must be tweened. Parameters that are optionally tweened, such as the angle for the slide effect, are set to a single value by default. In order to set an optionally-tweened parameter to a tweened value, the user must hold down the Option key when selecting the effect in the dialog box.

An application can provide its own user interface for entering multiple tween values for a parameter and choosing an appropriate tweener to perform interpolation, if required.

Note: Effect component authors do not need to write code to handle all the possible combinations of tween record types, as the details of the tween record are handled by using the standard QuickTime tweening APIs.

For more details on specifying which parameter values can contain tween values, see [Parameter Atom Type and ID](#) (page 78). For more details on supporting tweened parameters in your effect component, see [Tweening Parameter Values](#) (page 66).

Refer to [The Parameter Description Format](#) (page 78) for a complete description of the possible parameter descriptions you can place in your 'atoms' resource.

Slide

```
kSlideTransitionType('slid')
```

In a slide effect, source B slides onto the screen to cover source A. The angle from which source B enters the frame is stored in a parameter, with 0 degrees being the top of the screen.

The slide effect takes a maximum of two sources and has two parameters.

Use the descriptions below to help you understand what the parameters do. To learn how to use parameter atoms, see [Adding Video Effects to a QuickTime Movie](#) (page 11).

Name	Code	QTAtom Type	Description
Percentage	'pcnt'	kParameterType-DataFixed Always a tween	This parameter contains a tween. As the effect progresses, QuickTime renders the frame of the effect indicated by the tween's current value, as a percentage of the whole effect. For example, if the tween goes from 0 to 100, the effect renders completely; if the tween goes from 25 to 75, rendering begins 25% into the effect and terminates 75% through the effect.
Slide angle	'angl'	kParameterType-DataFixed Can be a tween	The angle from which source B will enter the frame. This value is expressed in degrees, with 0 degrees defined as the top of the screen.

Parameter Descriptions

Each effect component supplies a **parameter description** data structure that describes in detail the set of parameters that the effect has.

This section describes the parameter description format in detail. You need this information if you are writing an effect component. If you are writing an effect component, you should provide a parameter description as part of an 'atoms' resource (see [Supplying Parameter Description Information](#) (page 61) for more details).

You may also need this information if you are writing an application that presents its own user interface for setting effect parameters. In this case, you will need to parse parameter descriptions to generate appropriate controls to set parameter values. Most applications can simply use the `QTCreateStandardParameterDialog` function, and do not need to parse effect parameter descriptions.

Any software that uses an effect can request its parameter description. Typically, the parameter description is then passed to `QTCreateStandardParameterDialog` or especially, `ImageCodecCreateStandardParameterDialog`. These functions use the parameter description to display a user interface that allows users to choose the values of the parameters.

You are free to use the information in an effect's parameter description in other ways. For example, your application can use the default value atoms to construct an effect description.

Parameter descriptions are stored in a `QTAtomContainer` structure, and an application retrieves an effect's description by calling `ImageCodecGetParameterList`. This function takes a component instance and returns the parameter description for that component.

The code shown in Listing 4-15 opens the component specified in the variable `subType`. The code sets up the component description, then finds and opens the requested component. It then calls the `ImageCodecGetParameterList` function to fill out the parameter description for this effect.

Listing 4-15 Opening the image decompressor component

```
{
    // Set up a component description
    cd.componentType = 'imdc';           // Effects are image decompressor
                                        // components
    cd.componentSubType = subType;      // This is the name of the effect
                                        //(e.g. 'smpt')

    cd.componentManufacturer = 0;
    cd.componentFlags = 0;
    cd.componentFlagsMask = 0;
    // Find the required component. If it can't be found, generate an
    // error
    if ((theComponent = FindNextComponent(theComponent, &cd))==0)
    {
        err = paramErr;
        goto bail;
    }
    // Open the component
    gCompInstance = OpenComponent(theComponent);
    // Get the parameter description for the effect
    ImageCodecGetParameterList(gCompInstance, &parameterDescription);
}
```

An application can parse the returned parameter description using the standard QuickTime APIs that query `QTAtomContainer` data structures. This can be useful if you are writing an application that creates its own interface for users to customize effects.

This section describes the general format of the data returned in a parameter description.

Component-Defined Functions

This section defines the effect-specific functions that you may supply in your effect components. This section is only of interest to developers who are creating their own effects components; if you are writing an application that uses QuickTime video effects, you can skip this section.

The functions defined in this section are those called by the Component Manager through your component's dispatch function (see [What Effects Components Do](#) (page 59)).

These functions include

- [MyEffectSetup](#) (page 91)
- [MyEffectBegin](#) (page 92)
- [MyEffectRenderFrame](#) (page 92)
- [MyEffectCancel](#) (page 93)
- [MyEffectGetCodecInfo](#) (page 93)
- [MyEffectGetParameterListHandle](#) (page 94)
- [MyEffectGetSpeed](#) (page 94)
- [MyEffectValidateParameters](#) (page 95)

Note: The interfaces to these functions are described assuming you are using Apple's component dispatch helper code and the sample effect framework as described in [The Sample Effect Component](#) (page 71). Apple strongly recommends that you re-use these code samples where possible when implementing your own effect components.

If you are using the sample effect component, you can use the default implementations of several of these functions in most circumstances.

MyEffectSetup

The Component Manager calls this function when a sequence of frames is about to be rendered.

```
ComponentResult MyEffectSetup (
    EffectGlobals *glob,
    CodecDecompressParams *decompressParams);
```

Term	Definition
glob	A pointer to the effect's global data structure.
decompressParams	Information about the sequence that is about to be decompressed.

This function is called immediately before a client application such as `MoviePlayer` calls your component to render a sequence of frames.

Your component should examine the `capabilities` field of the `decompressParams` data structure to ensure that it can meet the requirements for executing this sequence. In particular, it should check the bit depth and pixel format requirements of the sequence. If the sequence requires a bit depth and pixel format combination that your component does not support, this function should return the nearest supported combination in the `decompressParams->capabilities` field. In this case, `QuickTime` will redirect all source and destination bitmaps through offscreen graphics worlds that have the bit depth and pixel format characteristics that you specify.

MyEffectBegin

The Component Manager calls this function to request that your component prepare to render a single frame of its effect.

```
ComponentResult MyEffectBegin (
    EffectGlobals *glob,
    CodecDecompressParams *decompressParams,
    EffectsFrameParamsPtr effect);
```

Term	Definition
glob	A pointer to the effect's global data structure.
decompressParams	Information about the current sequence of frames.
effect	The parameters describing this frame.

This function is called immediately before your `MyEffectRenderFrame` function. Your `MyEffectBegin` function should ensure that the information it holds about the current source and destination buffers and the parameter values for the effect are valid. If any of these have changed since the last call to `MyEffectBegin`, the new values should be read from the appropriate data structures.

This function is guaranteed to be called synchronously. In particular, this means you can allocate and move memory, and can call functions that allocate or move memory.

MyEffectRenderFrame


The Component Manager calls this function to request that your component render a single frame of its effect.

```
ComponentResult MyEffectRenderFrame (
    EffectGlobals *glob,
```

```
EffectsFrameParamsPtr effect);
```

Term	Definition
glob	A pointer to the effect's global data structure.
effect	The parameters describing this frame.

This function is called by a client application when your effect component needs to render a single frame of your effect. This function contains the implementation of your effect.

 **Warning:** This function is *not* guaranteed to be called synchronously. This means your function implementation must not allocate or move memory, or call any function that allocates or moves memory, in response to this call.

MyEffectCancel

The Component Manager calls this function to stop processing of the current effect.

```
ComponentResult MyEffectCancel (
    EffectGlobals *glob,
    EffectsFrameParamsPtr effect);
```

Term	Definition
glob	A pointer to the effect's global data structure.
effect	The parameters describing this frame.

This function is called by a client application (which may be QuickTime) to halt the rendering of the current sequence of frames before the last frame has been rendered. If your component is running synchronously, it should simply return `noErr`; no further calls to your `MyEffectRenderFrame` function will be made for this sequence.

If your component is running asynchronously, this function should dequeue all outstanding render frame requests, then return `noErr`.

MyEffectGetCodecInfo

The Component Manager calls this function to request information about the component.

```
ComponentResult MyEffectGetCodecInfo (
    EffectGlobals *glob,
    CodecInfo *info);
```

Term	Definition
glob	A pointer to the effect's global data structure.

Term	Definition
info	A pointer to the data structure that will contain the codec information.

This function is called by a client application (which may be QuickTime) to request information about your effect component. Your function should fill out the `CodecInfo` data structure passed to it. You can use the `GetComponentResource` function to retrieve a 'cdci' resource that stores this information if you have provided one in your component.

noErr	0	The function successfully filled out the info field.
paramErr	-50	Your function should return this value if the info parameter contains nil.

MyEffectGetParameterListHandle

The Component Manager calls this function to request a parameter description for this component.

```
ComponentResult MyEffectGetParameterListHandle (
    EffectGlobals *glob,
    Handle theHandle);
```

Term	Definition
glob	A pointer to the effect's global data structure.
theHandle	A pointer to a handle that will contain the parameter description of this effect.

This function is called by a client application (which may be QuickTime) to request a parameter description for your effect. This function can use the `GetComponentResource` function to retrieve an 'atms' resource that stores this information if you have provided one in your component.

MyEffectGetSpeed

The Component Manager calls this function to request information about the rendering speed of this effect component.

```
long MyEffectGetSpeed (
    EffectGlobals *glob,
    QTAtomContainer parameters,
    Fixed *pFPS)
```

Term	Definition
glob	A pointer to the effect's global data structure.
parameters	The current parameter values for this effect.
pFPS	A pointer to a <code>Fixed</code> value that will contain the rendering speed of this effect on exit.

This function is called by a client application (which may be QuickTime) to request information about the rendering speed of your effect. This function should return a `Fixed` value in FPS, which represents the rendering speed in frames-per-second of the effect.

If your effect can render in real time, it should return a value of `effectIsRealtime`. Otherwise, you should return an estimate of the number of frames your effect can render per second. Because rendering speeds are hardware-dependent, effect authors can choose to measure actual rendering speeds in this function. Alternatively, effect authors can choose to return a single value for all hardware configurations, estimating the value for a reference hardware platform.

Apple recommends that the values returned are rounded down to the nearest common frames-per-second value, such as 15, 24 or 30.

MyEffectValidateParameters

If your effect implements this optional function, the Component Manager calls it whenever the user changes a parameter value in the standard parameter dialog box, or attempts to dismiss the dialog.

```
ComponentResult MyEffectValidateParameters (
    EffectGlobals *glob QAtomContainer parameters,
    QTParameterValidationOptions validationFlags,
    StringPtr errorString);
```

Term	Definition
<code>glob</code>	A pointer to the effect's global data structure.
<code>parameters</code>	The current parameter values for this effect.
<code>validationFlags</code>	Flags that indicate whether a parameter value has changed or the user is dismissing the standard parameter dialog box.
<code>errorString</code>	A <code>StringPtr</code> that contains an error string explaining to the user why the validation has failed.

This optional function is called by a client application (which may be QuickTime) when your effect's standard parameter dialog box is being displayed. It can be called in two circumstances: if the user changes a parameter value in the dialog box; or if the user dismisses the dialog box by clicking OK.

The purpose of this function is to allow your effect to validate its parameters. The current parameter values are passed to the effect in `parameters`. If all of these values are valid, this function should return `noErr`. Otherwise, you should return a `paramErr` and put an explanatory message in the `errorString` parameter.

Video Effects API

Introduction

This chapter describes the constants, data types, and functions defined in QuickTime that support video effects.

Constants

This section describes the constants defined in QuickTime to support video effects.

- [Effects List Atom Names](#) (page 97)
- [Effect Action Selectors](#) (page 98)
- [Get Options for QTGetEffectsList](#) (page 99)
- [Standard Parameter Dialog Box Options](#) (page 99)
- [ImageCodecValidateParameters Options](#) (page 99)
- [Effect Speed Flag](#) (page 100)

Effects List Atom Names

These constants specify the four character codes for the two atom types in the atom container returned by calls to `QTGetEffectsList`.

```
enum {
    kEffectNameAtom = FOUR_CHAR_CODE('name'),
    kEffectTypeAtom = FOUR_CHAR_CODE('type')
}
```

Term	Definition
<code>kEffectNameAtom</code>	The code for the atom containing the name of an entry in the effects list.
<code>kEffectTypeAtom</code>	The code for the atom containing the type of an entry in the effects list.

Effect Action Selectors

These constants specify the action selectors you can pass to the functions

`QTStandardParameterDialogDoAction` and `ImageCodecStandardParameterDialogDoAction`.

```
enum {
    pdActionConfirmDialog,
    pdActionSetAppleMenu,
    pdActionSetEditMenu,
    pdActionGetDialogValues,
    pdActionSetPreviewUserItem,
    pdActionSetPreviewPicture,
    pdActionSetColorPickerEventProc,
    pdActionSetDialogTitle,
    pdActionGetSubPanelMenu,
    pdActionActivateSubPanel,
    pdActionConductStopAlert
}
```

Term	Definition
<code>pdActionConfirmDialog</code>	Retrieves the current parameter values from the standard parameters dialog box. The parameter values are placed in the <code>parameters</code> parameter that was passed to the function that created the dialog box.
<code>pdActionSetAppleMenu</code>	Passes the menu handle for the current Apple menu to the standard parameters dialog box.
<code>pdActionSetEditMenu</code>	Passes the menu handle for your application's Edit menu to the standard parameters dialog box.
<code>pdActionGetDialogValues</code>	Retrieves the current parameter values from the standard parameters dialog box. This parameter values are placed in the <code>params</code> parameter of the <code>QTStandardParameterDialogDoAction</code> or <code>ImageCodecStandardParameterDialogDoAction</code> function.
<code>pdActionSetPreviewUserItem</code>	Passes the number of the user item resource that should be replaced by the effect preview movie clip.
<code>pdActionSetPreviewPicture</code>	Sets the images that are used in the preview window of the standard parameters dialog box.
<code>pdActionSetColorPickerEventProc</code>	Sets the function that will be used when the standard parameters dialog box needs to display a color picker control.
<code>pdActionSetDialogTitle</code>	Sets the title of the standard parameters dialog box.
<code>pdActionGetSubPanelMenu</code>	Returns a <code>menuHandle</code> containing the pop-up menu used to switch between multiple parameter panels. You can parse this data structure to derive a list of the panels that are being used by the standard parameters dialog box.
<code>pdActionActivateSubPanel</code>	Sets the currently active panel of the standard parameter dialog box.
<code>pdActionConductStopAlert</code>	Displays a Stop Alert containing a message.

Get Options for QTGetEffectsList

These constants define the flags that can be passed in the `getOptions` parameter of the `QTGetEffectsList` function.

```
enum {
    e1OptionsIncludeNoneInList
}
```

Term	Definition
<code>e1OptionsIncludeNoneInList</code>	Includes the "none" effect in the list of effects returned by <code>QTGetEffectsList</code> .

Standard Parameter Dialog Box Options

These constants are used to control how parameter values are entered into a standard parameters dialog box that is generated when you call the function `ImageCodecCreateStandardParameterDialog`.

```
enum {
    pdOptionsCollectOneValue,
    pdOptionsAllowOptionalInterpolations
}
```

Parameters that are flagged as `kAtomInterpolateIsOptional` in their parameter description can contain single values or tweens. These constants are used to specify that the dialog box should allow entry of either a single value or a pair of values for such parameters.

Term	Definition
<code>pdOptionsCollectOneValue</code>	This value indicates that only one value should be entered for parameters that can optionally be tweened.
<code>pdOptionsAllowOptionalInterpolations</code>	This value indicates that optionally-tweened parameters should be displayed with a user interface that allows the entry of tweened values.

ImageCodecValidateParameters Options

These are the values that can be passed in the `validationFlags` parameter of a call to `ImageCodecValidateParameters`.

```
enum {
    kParameterValidationNoFlags,
    kParameterValidationFinalValidation
}
```

Term	Definition
<code>kParameterValidation-NoFlags</code>	This value indicates that a standard validation should take place. <code>ImageCodecValidateParameters</code> is being called because the user has changed the value of a parameter in the standard parameters dialog box.
<code>kParameterValidation-FinalValidation</code>	This value indicates that this validation is the final validation before the standard parameters dialog box is dismissed. This is useful if you want to perform a single validation of the parameters just after the user clicks the OK button to dismiss the dialog box.

Effect Speed Flag

This is the value that an effect should return when `QTGetEffectSpeed` is called and the component can run the effect in real time.

```
enum {
    effectIsRealtime
}
```

Term	Definition
<code>effectIsRealtime</code>	The effect can process frames in real time.

Data Types

This section describes the data types that are relevant to calling and creating effects.

The section describes the data types defined in QuickTime to support video effects.

- [Parameter Dialog Box Preview Image Specifier](#) (page 100)
- [Effect Source Descriptors](#) (page 101)
- [Effect Frame Description](#) (page 102)
- [The Decompression Parameters Structure](#) (page 102)

Parameter Dialog Box Preview Image Specifier

This data structure contains a picture that will be used as one of the preview images in the preview window of the standard parameters dialog box. The preview window shows previews of the effects the user chooses in the dialog box. QuickTime provides a default images, but you can replace one or more of these by calling `QTStandardParameterDialogDoAction` (or its low-level equivalent, `ImageCodecStandardParameterDialogDoAction`) with a `pdActionSetPreviewPicture` action selector.

The `sourceID` numbers correspond to the sources an effect uses. For example, an effect that uses one source will use the preview image with `sourceID` set to 1, while a two source effect will use preview pictures 1 and 2.

```
struct QTParamPreviewRecord {
    long        sourceID;
    PicHandle   sourcePicture;
};
typedef struct QTParamPreviewRecord QTParamPreviewRecord;
typedef QTParamPreviewRecord *QTParamPreviewPtr;
```

Term	Definition
<code>sourceID</code>	The number of the preview image.
<code>sourcePicture</code>	The preview image itself.

Effect Source Descriptors

These data structures describe the sources to an effect. The `SourceData` data structure contains a pointer to raw image compression manager image data, if the effect is being executed outside of a QuickTime movie, or to an effect that acts as the source, if the effect is being executed as part of an effect track in a QuickTime movie. The `EffectSourcePtr` data structure holds information about the type of source, as well as pointers to the track data of the effect and to the next source in the input chain.

```
typedef struct EffectSource    EffectSource;
typedef EffectSource          *EffectSourcePtr;

union SourceData {
    CDSequenceDataSourcePtr image;
    EffectSourcePtr        effect;
};
typedef union SourceData SourceData;
```

Term	Definition
<code>image</code>	A pointer to the raw source image data.
<code>effect</code>	A pointer to the effect.

```
struct EffectSource {
    long        effectType;
    Ptr        data;
    SourceData source;
    EffectSourcePtr next;
};
```

Term	Definition
<code>effectType</code>	The type of the effect or the constant <code>kEffectRawSource</code> if the source is raw image compression manager data.

Term	Definition
data	A pointer to the track data for the effect.
source	The source itself.
next	A pointer to the next source in the input chain.

Effect Frame Description

This data structure contains the parameters of a single frame of an effect. It is passed to the `MyEffectBegin` and `MyEffectRenderFrame` functions to describe the frame to be rendered.

```
struct EffectsFrameParams {
    ICMFrameTimeRecord  frameTime;
    long                effectDuration;
    Boolean              doAsync;
    unsigned char       pad[3];
    EffectSourcePtr     source;
    void *               refCon;
};
typedef struct EffectsFrameParams EffectsFrameParams;
typedef EffectsFrameParams *EffectsFrameParamsPtr;
```

Term	Definition
frameTime	Timing data for the current frame. This record includes information such as the total number of frames being rendered in this sequence, and the current frame number.
effectDuration	The duration of a single effect frame.
doAsync	This field contains <code>true</code> if the effect can process asynchronously.
source	A pointer to the input sources.
refCon	A pointer to storage for this instantiation of the effect.

The Decompression Parameters Structure

Several fields of the decompression parameters structure (`CodecDecompressParams`) are relevant to effects, and in particular are used when you are writing your own effect component. This section describes only those fields of this data structure that apply to effects.

```
struct CodecDecompressParams {
    ...
    CodecCapabilities      *capabilities;
    ...
    ICMFrameTimePtr       frameTime;
    ...
    UInt16                 majorSourceChangeSeed;
    UInt16                 minorSourceChangeSeed;
};
```

```

    CDSequenceDataSourcePtr    sourceData;
    ...
    OSType                    **wantedDestinationPixelTypes;
    ...
    Boolean                    needUpdateOnTimeChange;
    ...
    Boolean                    needUpdateOnSourceChange;
    ...
    long                       requestedBufferWidth;
    long                       requestedBufferHeight;
};
typedef struct CodecDecompressParams CodecDecompressParams;

```

Term	Definition
capabilities	A pointer to a <code>CodecCapabilities</code> data structure containing the required capabilities of the effect.
frameTime	The current frame time. The information in this parameter can be used to calculate the current frame number, relative to the total duration of the effect.
majorSource-ChangeSeed	An integer value that is incremented each time a data source is added or removed. This provides a fast way for an effect component to know when its data sources have changed.
minorSource-ChangeSeed	An integer value that is incremented each time a data source is added or removed, or the data contained in any of the sources changes. This provides a fast way for an effect component to know when a source changes.
sourceData	A pointer to a <code>CDSequenceDataSource</code> structure that contains the head of a linked list of all data sources. Because each data source contains a link to the next data source, an effect component can access all data sources from this field.
wantedDestination-PixelTypes	A handle to a zero-terminated list of non-RGB pixel formats that the component can decompress. Set this value to nil if your component does not support non-RGB pixel spaces. If your effect component supports non-RGB pixel formats, your component's <code>MyEffectSetup</code> function should place the list of them into this field. The Image Compression Manager (ICM) copies this data structure, so it is up to your component to dispose of it later. It is suggested that components allocate this handle during the <code>Open</code> routine and dispose of it during the <code>Close</code> routine.
needUpdateOn-TimeChange	Setting this field to true indicates to the ICM that your component's <code>EffectBegin</code> and <code>EffectRenderFrame</code> functions should be called whenever the time of the movie changes. The default value of this field is true.
needUpdateOnSource-Change	Setting this field to true indicates to the ICM that your component's <code>EffectBegin</code> and <code>EffectRenderFrame</code> functions should be called whenever one or more of the sources to the effect changes. The default value of this field is false.
requestedBufferWidth	Specifies the width of the image buffer to use, in pixels. For this value to be used, the <code>codecWantsSpecialScaling</code> flag in the <code>CodecCapabilities</code> record must be set.

Term	Definition
requestedBuffer-Height	Specifies the height of the image buffer to use, in pixels. For this value to be used, the <code>codecWantsSpecialScaling</code> flag in the <code>CodecCapabilities</code> record must be set.

Functions

This section lists the functions supported by effects components. Applications developers will typically call the high-level functions directly.

■ High-Level Functions

- ❑ `QTGetEffectsList`
- ❑ `QTCreatStandardParameterDialog`
- ❑ `QTIStandardParameterDialogEvent`
- ❑ `QTDissmissStandardParameterDialog`
- ❑ `QTStandardParameterDialogDoAction`
- ❑ `QTGetEffectSpeed`

■ Low-level Functions

- ❑ `ImageCodecGetParameterList`
- ❑ `ImageCodeCreatStandardParameterDialog`
- ❑ `ImageCodecIsStandardParameterDialogEvent`
- ❑ `ImageCodecStandardParameterDialogDoAction`
- ❑ `ImageCodecDissmissStandardParameterDialog`

■ Utility Functions

- ❑ `MakeImageDescriptionForEffect`

Document Revision History

This table describes the changes to *QuickTime Video Effects and Transitions Guide*.

Date	Notes
2007-05-03	Updated illustrations.
2006-01-10	Removed obsolete material and changed title from "Filters, Effects, and Transitions."
2002-09-17	New document that describes QuickTime video effect components and explains how to use them.

REVISION HISTORY

Document Revision History