




QuickTime Wired Movies And Sprite Animation

A Guide For Content Authors and Tool Developers



Apple Computer, Inc.
Technical Publications
July, 1999

 Apple Computer, Inc.
© 1999 Apple Computer, Inc.
All rights reserved.

No part of this publication or the software described in it 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 Computer, Inc., except in the normal use of the software or to make a backup copy of the software or documentation. The same proprietary and copyright notices must be affixed to any permitted copies as were affixed to the original. This exception does not allow copies to be made for others, whether or not sold, but all of the material purchased (with all backup copies) may be sold, given, or loaned to another person. Under the law, copying includes translating into another language or format. You may use the software on any computer owned by you, but extra copies cannot be made for this purpose.

The Apple logo is a trademark of Apple Computer, 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 book. Apple retains all intellectual property rights associated with the technology described in this book. Every effort has been made to ensure that the information in this manual is accurate. Apple is not responsible for printing or clerical errors.

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

Apple, the Apple logo, FireWire, Mac, Macintosh, and QuickTime are trademarks of Apple Computer, Inc., registered in the United States and other countries.

The QuickTime logo is a trademark of Apple Computer, Inc.

Adobe, Acrobat, Photoshop, and PostScript are trademarks of Adobe Systems Incorporated or its subsidiaries and may be registered in certain jurisdictions.

MacPaint is a trademark of Apple Computer, Inc., registered in the U.S. and other countries.

Helvetica and Palatino are registered trademarks of Linotype-Hell AG and/or its subsidiaries.

Indeo and Intel are registered trademarks of Intel.

ITC Zapf Dingbats is a registered trademark of International Typeface Corporation.

PowerPC and the PowerPC logo are trademarks of International Business Machines Corporation, used under license therefrom.

Simultaneously published in the United States and Canada.

Printed in the United States of America.

LIMITED WARRANTY ON MEDIA AND REPLACEMENT

ALL IMPLIED WARRANTIES ON THIS MANUAL, INCLUDING IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED IN DURATION TO NINETY (90) DAYS FROM THE DATE OF DISTRIBUTION OF THIS PRODUCT.

Even though Apple has reviewed this manual, APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS MANUAL, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS MANUAL IS DISTRIBUTED “AS IS,” AND YOU 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 MANUAL, 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

Preface	About This Book	9
	Book Structure	9
	Conventions Used in This Book	10
	Special Fonts	10
	Types of Notes	11
	Development Environment	11
Chapter 1	Introduction to Wired Movies, Sprites, and the Sprite Toolbox	13
	Introduction to Wired Movies	13
	Adding Actions	14
	QuickTime Events	14
	Actions and Their Targets	15
	Action Parameters	16
	Expressions	17
	Operators	17
	Operands	17
	External Movie Targets for Wired Actions in QuickTime 4	18
	Introduction to Sprite Animation and the Sprite Toolbox	19
	Sprites and the Sprite Toolbox	19
	Sprite World Characteristics	20
	Sprite Tracks	21
	Sprite Animation	22
	Sprite Spatial Concepts	22
	Sprite Track's Local Coordinate System	23
	Source Box	23
	Sprite's Bounding Box	24
	Sprites' Four Corners	25
	Registration Points	25
	Display Space	31
	Sprite Properties	32
	Defining QuickTime Atoms and Atom Containers	33

Chapter 2 The Sprite Media Handler 35

About the Sprite Media Handler	35
Hit Testing Flags	36
Key Frame Samples and Override Samples	37
Sprite Track Media Format	38
Assigning Group IDs	40
Sprite Image Registration	41
Sprite Track Properties	42
Alternate Sources for Sprite Image Data	44
Supported Modifier Inputs	45
New Features in QuickTime 4	46
Referenced Sprite Images	46
Sprite Button Behaviors	46
Action Handler Property	47
String Variable Support	47

Chapter 3 Authoring Wired Movies and Sprite Animations 49

Authoring Movies with the Sprite Media Handler	49
Defining a Key Frame Sample	50
Creating the Movie, Sprite Track, and Media	50
Adding Images to the Key Frame Sample	51
Adding More Images for Other Sprites	52
Adding Sprites to the Key Frame Sample	53
Adding More Actions to Other Sprites	57
Adding Sample Data in Compressed Form	58
Defining Override Samples	58
Setting Properties of the Sprite Track	60
Getting Sprite Data From a Modifier Track	61
Authoring Wired Movies	65
Actions of the First Penguin	67
Actions of the Second Penguin	68
Creating a Wired Sprite Movie	68
Assigning the No Controller to the Movie	68
Setting Up the Sprite Track's Properties	69
Adding an Event Handler to the Penguin	70
Adding a Series of Actions to the Penguins	71

Important Things to Note in the Sample Code	72
New Authoring Features in QuickTime 4	73
Specifying an External Movie Target for an Action	73
Tagging a Movie with a Name or ID	74
Naming a Movie	74
Setting a Movie's ID	74
Supporting External Movie Targets in Your Application	74
New Grammar for Specifying String Parameters to Actions and Operands	75
Wiring a Movie by Adding an HREF Track	76
About HREFTracks	76
HREFTrack Syntax	77
Creating an HREFTrack	78

Chapter 4 Using the Sprite Toolbox to Create Sprite Animations 81

How To Add Sprite-Based Animations to an Application	81
Creating and Initializing a Sprite World	82
Creating and Initializing Sprites	84
Creating Sprites for a Sample Application	84
Animating Sprites	87
Disposing of a Sprite Animation	89
Sprite Hit Testing	90
Hit Testing Flags	91
Enhancing Sprite Animation Performance	91

Chapter 5 Flash Media Handler 93

Support For Macromedia's Flash in QuickTime 4	93
The Flash Media Handler	93
How To Add Wired Actions To a Flash Track	94
Extending the SWF Format	94
What You Need to Modify	95

Constants	99
Event Constants	99
Target Constants	101
Action Constants	103
Movie Action Constants	104
Actions for All Tracks	106
New Wired Movie Actions and Operands in QuickTime 4	107
New Actions	107
Actions for All Tracks	107
Actions That Do Not Have a Target	108
Actions for Sprite Tracks	111
Actions for QD3D Named Objects	112
Actions for Flash Tracks	113
New Operands	114
Actions for Spatial Tracks	117
Actions for Sound Tracks	117
Actions for Sprites in a Sprite Track	118
Actions for QuickTime VR Tracks	120
Actions for Music Tracks	121
Actions for Sprite Tracks	123
Actions That Do Not Have a Target	123
Control Statement Actions	126
Action Parameter Constants	127
Expression-Related Constants	130
Operator Type Constants	131
Operand Type Constants	133
Operands for a Movie	133
Operands for Any Tracks	134
Operands Targeting Spatial Tracks	134
Operands Targeting Sound Tracks	134
Operands for Sprites in a Sprite Track	135
Operands for a Sprite Track	137
Operands for a QuickTime VR Track	138
Operands That Have No Target	138
Additions to the Standard Movie Controller	140
Data Types	141

Chapter 7 Sprite Toolbox API Reference 143

Constants	143
Background Sprites	143
Flags for Sprite Hit Testing	143
Sprite Properties	144
Flags for SpriteWorldIdle	145
Sprite and Sprite World Identifiers	146
Sprite Toolbox Functions	147
Sprite World Functions	147
Sprite Functions	154

Chapter 8 Sprite Media Handler API Reference 161

Constants	161
Action Media Format Atoms	161
Action Sprites Media Format Extensions	161
Sprite Track Property Atoms	162
Atom Types	163
New Atom Types in QuickTime 4	164
Sprite Track Formats	165
Sprite Media Atom Types	165
Sprite Media Handler Functions	169
Sprites Functions Specific to Wired Sprites	179
SpriteDescription Structure	180

Appendix A QTAAtomContainer-Based Data Structure Descriptions 181

QTAAtomContainer Description Key	181
----------------------------------	-----

Appendix B Sprite Media Handler Media Format Definition 183

Appendix C QTWiredSprite.c Sample Code 195

About This Book

This book is a developer's guide to QuickTime 4 wired movies, sprites, and sprite animation for Macintosh and Windows. It is aimed at content authors and tool developers who want to master the fundamentals of this new Apple technology and build their own applications using QuickTime. This book supersedes all existing documentation, including its predecessor, "Programming with QuickTime Sprites."

The original QuickTime-related *Inside Macintosh* books documented QuickTime for Macintosh through the 1.5 software release. If you are an application developer, your main source for information on programming in QuickTime is now the combination of this book, along with the suite of other QuickTime-related books, including *QuickTime 4 Reference*, *QuickTime Music Architecture*, *Inside Macintosh: QuickTime*, *Inside Macintosh: QuickTime Components*, and *Mac OS For QuickTime Programmers*.

Book Structure

Chapter 1, "Introduction to Wired Movies, Sprites, and the Sprite Toolbox," provides a general introduction to QuickTime wired movies, sprites, and the sprite toolbox. It also discusses some of the new wired movie features added to QuickTime 4.

Chapter 2, "The Sprite Media Handler," describes how you can use the sprite media handler to add a sprite animation track to a QuickTime movie. This media handler provides routines for manipulating the sprites and images in a sprite track.

Chapter 3, "Authoring Wired Movies and Sprite Animations," describes how you can author wired movies and sprite animations using the sprite media handler.

Chapter 4, "Using the Sprite Toolbox to Create Sprite Animations," discusses the sprite toolbox, which is a set of data types and functions you can use to add sprite-based animation to an application.

Chapter 5, “Flash Media Handler,” discusses the addition of a new Flash Media Handler and QuickTime 4 support for the interactive playback of Macromedia Flash SWF files, commonly played back using Macromedia’s ShockWave Plugin. This new feature allows an SWF file to be treated as a track within a QuickTime movie.

Chapter 6, “Wired Movies API Reference,” provides an extensive reference of constants and data types available to your application for wired movies support. The chapter also includes information about new atom types which have been added to QuickTime 4. In addition, it describes the new movie actions and operands available to your application in QuickTime 4.

Chapter 7, “Sprite Toolbox API Reference,” describes the constants and functions available to your application in the sprite toolbox. It also describes the functions provided by the Movie Toolbox for sprite support.

Chapter 8, “Sprite Media Handler API Reference,” provides a reference of constants and functions available to your application, using the sprite media handler. The chapter also describes the new atom types that have been added to QuickTime 4.

Appendix A provides you with a key to QTAtomContainer-based data structures that are being widely used in QuickTime. Appendix B describes formats and defines a grammar for sprite media handlers and their data containers. Appendix C presents sample code from `MakeActionSpriteMovie.c`, listed in the QuickTime 4 SDK, which allows you to create a sample wired sprite movie containing one sprite track.

Conventions Used in This Book

This book provides various conventions to present information. Words that require special treatment appear in specific fonts or font styles. Certain types of information, such as parameter blocks, use special fonts so that you can scan them quickly.

Special Fonts

All code listings, reserved words, and the names of actual data structures, constants, fields, parameters, and functions are shown in Letter Gothic (`this is Letter Gothic`).

Words that appear in **boldface** are key terms or concepts that are defined in the glossary.

Types of Notes

There are several types of notes used in this book.

Note

A note like this contains information that is interesting but not essential to an understanding of the main text. ♦

IMPORTANT

A note like this contains information that is essential for an understanding of the main text. ▲

▲ **WARNING**

A warning like this indicates potential problems that you should be aware of as you design your application. Failure to heed these warnings could result in system crashes or loss of data. ▲

Development Environment

The functions described in this book are available using C interfaces. How you access them depends on the development environment you are using.

Code listings in this book are shown in ANSI C. They suggest methods of using various functions and illustrate techniques for accomplishing particular tasks. Although most code listings have been compiled and tested, Apple Computer Inc., does not intend for you to use these code samples in your application.

P R E F A C E

Introduction to Wired Movies, Sprites, and the Sprite Toolbox

This chapter provides you with a general introduction to QuickTime wired movies, sprites, and the sprite toolbox. It also discusses some of the new wired movie features added to QuickTime 4.

If you are a content author or tool developer, you'll want to read this chapter to grasp the fundamentals of how wired movies, sprites, and the sprite toolbox work together in the QuickTime software architecture.

Wired movies were originally introduced in QuickTime 3 and have been enhanced in QuickTime 4. These interactive movies can be played in any Web browser using the QuickTime plug-in, and in all applications as long as they use the QuickTime movie controller API.

This chapter is divided into the following major sections:

- “Introduction to Wired Movies” (page 13) introduces QuickTime events, actions, targets, parameters, expressions, operands, and operators.
- “Introduction to Sprite Animation and the Sprite Toolbox” (page 19) introduces you to the fundamentals of sprite animation and the sprite toolbox.
- “Defining QuickTime Atoms and Atom Containers” (page 33) discusses QuickTime atoms, which QuickTime uses to store most of its data using specialized structures in memory. Movies themselves are atoms, as are tracks, media, and data samples. An understanding of atoms is important in working with wired movies, sprites and sprite animation.

Introduction to Wired Movies

Wired movies enable you to create QuickTime movies that are interactive. User input is translated into QuickTime events. In response to these QuickTime

events, actions may be performed. Each action has a specific **target**, which is the element in a movie the action is performed on. Target types include sprites, tracks, and the movie itself. A few actions do not require a target. Actions have a set of parameters that help describe how the target element is changed.

Typical wired actions—such as jumping to a particular time in a movie or setting a sprite’s image index—enable you to create a sprite that acts as a button. In response to a mouse down event, for example, a wired sprite could change its own image index property, so that its button-pressed image is displayed. In response to a mouse up event, the sprite can change its image index property back to the button up image and, additionally, specify that the movie jump to a particular time.

Adding Actions

When you wire a sprite track, you add **actions** to it. Wired sprite tracks may be the only tracks in a movie, but they are commonly used in concert with other types of tracks. Actions associated with sprites in a sprite track, for example, can control the audio volume and balance of an audio track, or the graphics mode of a video track.

Wired sprite tracks may also be used to implement a graphical user interface for an application. Applications can find out when actions are executed, and respond however they wish. For example, a CD audio controller application could use an action sprite track to handle its graphics and user interface.

These wired actions are not only supported in sprite tracks. In principle, you can “wire” any QuickTime media handler. In QuickTime 4, all of these may contain actions: QuickDraw 3D, QuickTime VR, text, and sprites.

For a complete description of all available wired actions, including those new to QuickTime 4, refer to Chapter 6, “Wired Movies API Reference.”

QuickTime Events

When it is associated with a QuickTime movie, the movie controller checks to see if a track in the movie provides actions in response to QuickTime **events**. If one or more tracks provide actions, the movie controller will then monitor the activity of the mouse and send the appropriate mouse-related events.

These mouse events include: `kQTEventMouseClicked`, `kQTEventMouseClickedEnd`, `kQTEventMouseClickedEndTriggerButton`, `kQTEventMouseEnter`, **and** `kQTEventMouseExit`.

Other types of events do not require user interaction in order to be generated.

The `kQTEventIdle` event is sent to each sprite in a sprite track if that sprite track's `kSpriteTrackPropertyQTIdleEventsFrequency` property is set to a value other than the default value `kNoQTIdleEvents`.

The `kQTEventFrameLoaded` event is generated when a sprite track sample which contains actions for it is loaded.

Actions and Their Targets

Any number of actions may be executed in response to a single QuickTime event. By using sprite track variables to maintain state, as well as conditional and looping actions, more sophisticated event handlers may be created—similar to `If...Then` and `While` statements in the C programming language.

The target of an action specifies on which element of the movie the action should be performed. Each action has an associated target type. For example, the action `kActionSpriteSetVisible` should only target a sprite—not a track or movie. Most of the track actions need to target either a specific track type, or a subset of track types, such as spatial or audio tracks. The target types include sprite, track, movie, 3D nodes, and no target.

Sprite and track targets may be specified in several different ways using names, IDs, or indices.

Targets are resolved at the time their action is executed. This is important to keep in mind because movies may change over time. Actions which are intended to target a movie element for the current movie time should precede actions which change the movie time. Sprites, for example, may be considered to have a lifetime lasting from one key frame to the next, so a sprite with a certain ID at one time may be different from a sprite with the same ID at another time in the movie.

Note that you may only target “live” movie elements, that is, ones that exist for the current movie time.

All events except the `frameLoaded` event are sent to a specific sprite or QuickDraw 3D node. This sprite is considered the *current default Sprite target*,

and its track is considered the *current default Track target*. Since the `frameLoaded` event is sent to a sprite track, only the default track target is set.

Action Parameters

Actions have some number of required parameters. The parameters each have a data type. For example, the `SpriteSetVisible` action has a single `Boolean` parameter which makes the sprite visible if set to `true`, and invisible if set to `false`.

Parameters with numeric data types may optionally be specified by an expression. The `SpriteSetVisible` action, for example, could have an expression which evaluates to `true` if the movie is playing and `false` if it is stopped.

Options which modify a parameter's value may be specified for some parameters. Each action defines which options are allowed for its parameters. Parameters which allow options to be specified are typically associated with a property of the action's target. The current value of this property is used in conjunction with the parameter's value and options to determine the new value.

The `kActionFlagActionIsDelta` constant takes the current property value and adds the parameter's value to it. This value is pinned to the minimum and maximum values. The `kActionFlagParameterWrapsAround` constant causes the value to wrap within the range defined by the minimum and maximum value. If the new value is greater than the maximum value, it wraps around to the minimum, plus the difference between the new and the maximum value.

Parameters all have default minimum and maximum values. When using the delta, or delta with wraparound options, the minimum and maximum value options further limit this range.

The `kActionFlagActionIsToggle` constant is used with properties that only have two possible values, such as a visible property which may be either `true` or `false`. Using it repeatedly on a sprite's visible property, for example, will toggle it between visible and invisible. The actual value of the parameter is ignored when using the toggle. For more information about these options, see "Action Parameter Constants" (page 127).

Note

Named time parameters use the indexed chapter text tracks to obtain time values from the names. ♦

Expressions

Expressions may generally be used in place of numeric and Boolean values. Numeric action parameters, action target IDs, and action target indexes may all use expressions. They are also used in conjunction with the `Case` and `While` statement actions as conditional Boolean expressions.

Expressions may contain just a single operand, or may be complex, containing any number of operators and operands.

For details on the grammar of expressions, see Appendix B.

IMPORTANT

Expressions are evaluated internally as single-precision, floating-point numbers. This means that all operands with numeric data types that are used in an expression are cast to a single precision floating point. For a few of the operand types, such as the sprite ID operands, it is possible to have a round-off error problem. This can be avoided by using sprite IDs that can be expressed using single-precision, floating-point numbers. ▲

Operators

Operators are used in expressions, and are applied to their operands to calculate numeric values. The data format for Binary operations is prefix-based.

Binary operators may be applied to a list of two or more operands. They are first applied to the first two operands, then applied to this result and the next operand in the list. For example, $(2 * (4 * 6))$ can be represented as the `kOperatorMultiply` operator with a list of three `kOperandConstant` operands, containing the values 4, 6, and 2 in that order.

Unary operators are applied to a single operand.

Operands

Each operand is evaluated as part of an expression. Most operands have specific target types, similar to actions, since they evaluate to the current value of a specific property of the target. For example, the `kOperandQTVRPanAngle` returns the current pan angle of the operand's target QuickTime VR track.

Other operands, such as `kOperandKeyIsDown` and `kOperandMouseLocalHLoc`, allow for a polling form of input by determining the current state of the keyboard or the location of the mouse.

Constants may be specified using the `kOperandConstant` operand.

The `kOperandExpression` allows for expressions to be nested within other expressions.

External Movie Targets for Wired Actions in QuickTime 4

QuickTime 4 allows your application to target external movies in addition to tracks and objects within tracks, such as sprites and QuickDraw 3D nodes.

In QuickTime 4 wired actions may be performed on an element of another movie. For example, you can create a movie which acts as a custom Movie Controller, setting the rate and volume of one or more separate movies on the same Web page. Or you could create two co-operative, talking head movies that have a conversation, each waiting for the other to finish before speaking their next piece. Because each movie has its own independent time base, you can achieve results that are impossible if you used a single wired movie that uses this mechanism.

External movies may be referred to either by name or by ID. There is a standard way to tag a movie with name and ID properties.

Since a QuickTime movie has no knowledge of what other QuickTime movies are currently open, the software that is playing the movies works with a Movie Controller in order to resolve the external movie names or IDs to actual movie references. The versions of the QuickTime plug-in and QuickTime Player released with QuickTime 4 both support external movie targets. Using the QuickTime plug-in, the movies should all be on the same Web page. Using the Movie Player, they should all be open movie documents.

For more information about authoring movies with external targets, refer to “New Authoring Features in QuickTime 4” (page 73).

Introduction to Sprite Animation and the Sprite Toolbox

This section introduces you to the fundamentals of sprite animation and the sprite toolbox. Sprites were new to QuickTime 2.5 and have since been enhanced in QuickTime 3 and further enhanced in QuickTime 4.

This section also discusses the ways in which sprite animation differs from traditional video animation. The metaphor of a sprite animation as a theatrical play is used, in which **sprite tracks** are characterized as the boundaries of the stage and a **sprite world** as the stage itself. To extend the metaphor, you may want to think of **sprites** as actors performing on that stage.

Each sprite has properties that describe its location and appearance at a given time. During an animation sequence, the application modifies the sprite's properties to cause it to change its appearance and move around the screen. Sprites may be mixed with still-image graphics to produce a wide variety of effects while using relatively little memory.

You use the sprite toolbox to add sprite-based animation to your application. The sprite toolbox, which is a set of data types and functions, handles all the tasks necessary to compose and modify sprites, their backgrounds and properties, in addition to transferring the results to the screen or to an alternate destination.

Sprites and the Sprite Toolbox

This section introduces you to the terminology used to define sprites and describes the characteristics that govern the creation of sprite animation in an application.

If you're writing an application that uses sprite animation *outside* of a QuickTime movie, you use the routines available to you in the sprite toolbox. If your application is designed to work with QuickTime movies, you can take advantage of the routines available to you in the sprite media handler, which is discussed in Chapter 2, "The Sprite Media Handler."

The sprite toolbox is a set of data types and functions you can use to add sprite-based animation to an application. The sprite toolbox handles invalidating appropriate areas as sprite properties change, the composition of sprites and their background on an offscreen buffer, and the transfer of the result to the screen or to an alternate destination.

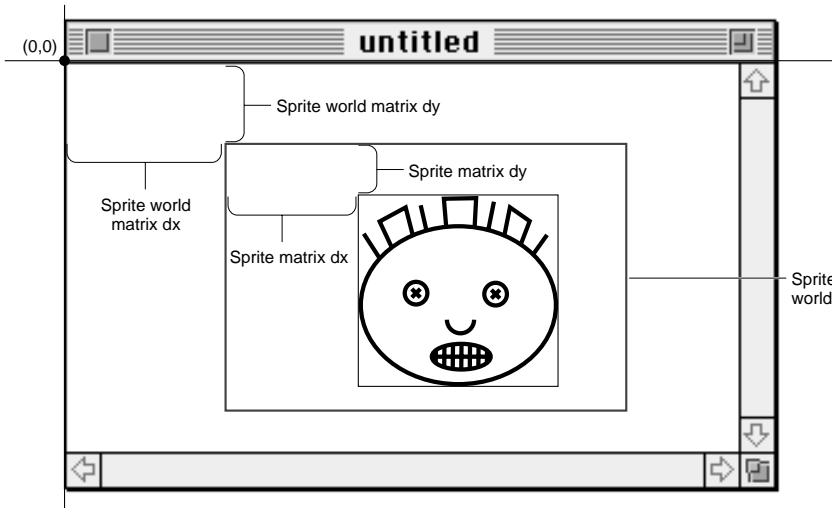
Sprite World Characteristics

A **sprite world** is a graphics world for a sprite animation. To create a sprite animation in an application, you must first create a sprite world. You do not need to create a sprite world to create a sprite track in a QuickTime movie.

Once you have created a sprite world, you create sprites associated with that sprite world. You can think of a sprite world as a stage on which your sprites perform. When you dispose of a sprite world, its associated sprites are disposed of as well.

For sprites in a sprite world, you modify a sprite's properties by calling the `SetSpriteProperty` function, passing a constant to indicate which property you want to modify. `SetSpriteProperty` invalidates the appropriate portions of the sprite world, which are redrawn when `SpriteWorldIdle` is called.

When you call `SetSpriteProperty` to modify a property of a sprite, `SetSpriteProperty` invalidates the appropriate regions of the sprite world. When your application calls `SpriteWorldIdle`, the sprite world redraws its invalid regions. A sprite's sprite world coordinate system is defined by translating the sprite's display coordinate system by the sprite world's matrix, as shown in Figure 1-1.

Figure 1-1 Sprite world coordinate system

For sprites in a sprite world, you control a sprite's image by setting the sprite's `kSpritePropertyImageDescription` and `kSpritePropertyImageDataPtr` properties.

Sprite Tracks

For sprites in a sprite track, all sprite images are stored in one of the sprite track's key frame samples. This allows the sprites in the sprite track to share images. A sprite's image index (`kSpritePropertyImageIndex`) specifies the sprite's current image in the pool of available images. All images assigned to a sprite must share the same image description, unless you assign group IDs (`kSpriteImagePropertyGroupID`).

For sprites in a sprite track, you modify a sprite property by creating an override sample of the appropriate type.

Three sprite track properties, `kSpriteTrackPropertyBackgroundColor`, `kSpriteTrackPropertyOffscreenBitDepth`, and `kSpriteTrackPropertySampleFormat`, describe properties of a sprite track in a QuickTime movie. These properties are discussed in more detail in Chapter 2, "The Sprite Media Handler."

Sprite Animation

Sprite animation differs substantially from traditional video animation. With traditional video animation, you describe a frame by specifying the color of each pixel. By contrast, with sprite animation, you describe a frame by specifying which sprites appear at various locations. At a given moment a sprite displays a single image selected from a pool of images shared by all of the sprites.

You can think of a sprite animation as a theatrical play. In a QuickTime movie, the sprite track bounds are the stage; in an application, a sprite world is the stage. The background is the play's set; the background may be a single solid color, an image, or a combination of images. The sprites are the actors in the play.

A sprite has **properties** that describe its location and appearance at a given point in time. During the course of an animation, you modify a sprite's properties to cause it to change its appearance and move around the set or stage.

Each sprite has a corresponding **image**. During the animation, you can change a sprite's image. For example, you can assign a series of images to a sprite in succession to perform cell-based animation.

Sprite Spatial Concepts

This section explains sprite spatial concepts, which you may need to understand in order to work with sprites both on the desktop (outside a QuickTime movie) and within QuickTime movies. These concepts include

- a sprite track's local coordinate system
- the source box of a sprite image
- bounding box
- a sprite's "four corners"
- registration points
- display space

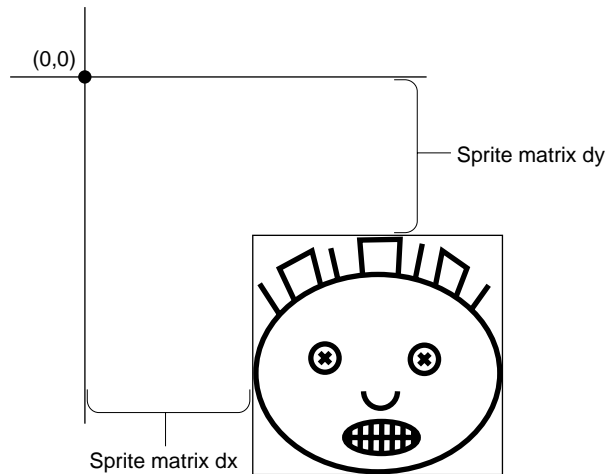
Sprite Track's Local Coordinate System

A sprite track's **local coordinate system**—where a sprite is displayed within a sprite world or a sprite track—is defined by the sprite's matrix and its image's registration point.

This is the coordinate system, shown in Figure 1-2, that results when the sprite track's matrix and the movie matrix are both ignored. The origin is the sprite track's upper left corner.

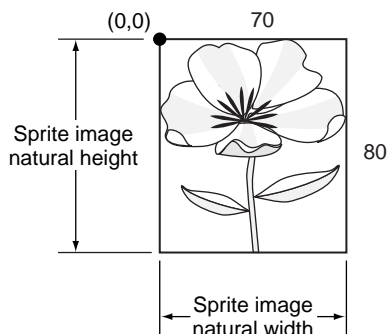
For example, if a sprite's matrix contains a horizontal translation of 50 and a vertical translation of 25, the sprite will be positioned such that its current image's registration point is located 50 pixels to the right of the sprite track's left side, and 25 pixels down from the top of the sprite track.

Figure 1-2 A track's local coordinate system of a sprite



Source Box

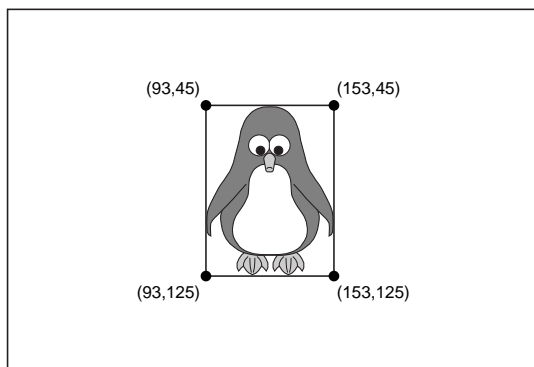
A sprite's **source box**, as shown in Figure 1-3, is defined as a rectangle with a top-left point of $(0, 0)$, and its width and height set to the width and height of the sprite's current image.

Figure 1-3 A sprite's source box

Sprite's Bounding Box

A sprite's bounding box and its four corners are both expressed in its track's local coordinate system.

The bounding box of a sprite, shown in Figure 1-4, is the smallest rectangle that encloses the sprite's area after its matrix is applied. If a sprite is only translated, its bounding box will have the same dimensions as its source box. However, if the sprite is rotated 45 degrees, the bounding box may be larger than its source box.

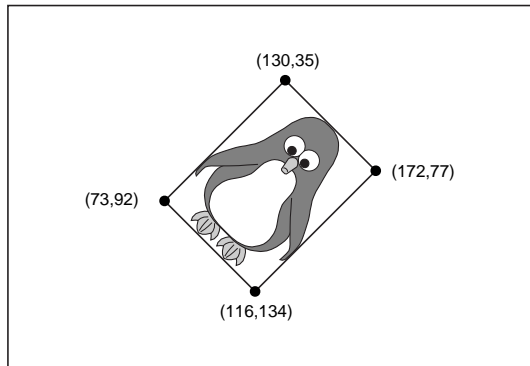
Figure 1-4 A bounding box in a sprite track's local coordinate system

Sprites' Four Corners

Some sprite actions and operands refer to a sprite's "four corners." These four corners are expressed in its track's local coordinate system. They are the points derived by taking the four corners of the sprite's source box and applying the current image's registration point and the sprite's source matrix. The first corner is the top-left, the second corner is the top-right, the third corner is the bottom-right, and the fourth corner is the bottom-left.

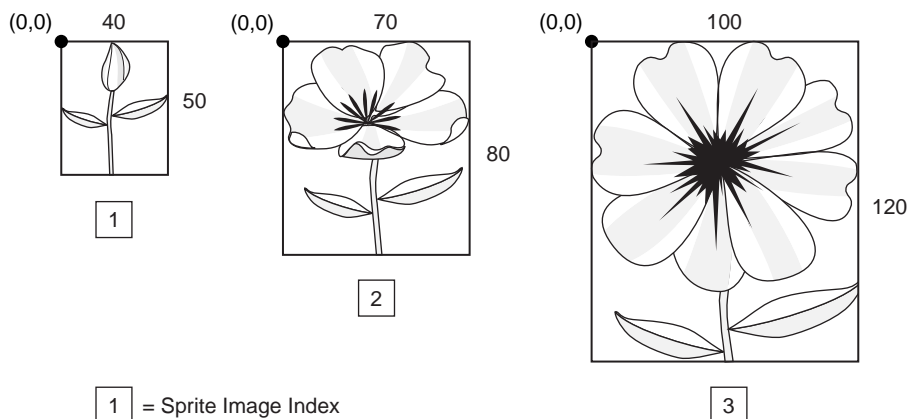
Figure 1-5 shows a rotated bounding box in a sprite track's local coordinate system.

Figure 1-5 The rotated bounding box becomes the sprite four corners



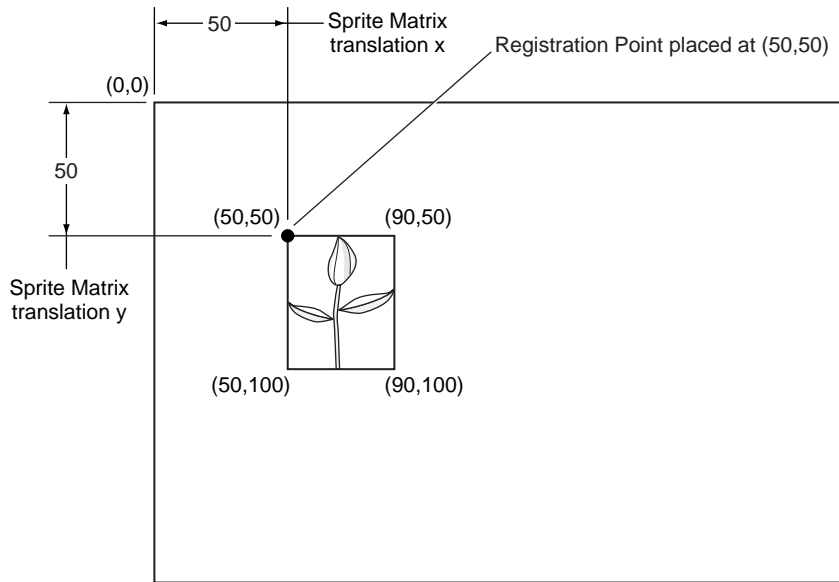
Registration Points

QuickTime 3 introduced sprite image **registration points**. These image registration points, shown in Figure 1-6, define an offset which is applied to a sprite's source matrix. A sprite's default registration point is (0,0), or the top left of its source box.

Figure 1-6 Default sprite image registration points

When a sprite with a default registration point of (0,0) is translated to a location by setting the x and y translation elements of its source matrix, the sprite's upper left corner is placed at the given location. If a sprite's source box is 100 pixels wide and 100 pixels tall, then setting the sprite image's registration point to (50,50) causes the center of the sprite to be translated to the x and y translation of its source matrix. This also causes the wired sprite action `kActionSpriteRotate` to rotate the sprite about its center.

Figure 1-7 shows a default registration point in a sprite track's local coordinate system.

Figure 1-7 Default registration point in a sprite track's local coordinate system

If your animation is cell-based, your images may vary in size, so you may want the registration for the center in order for the images used by the sprites to line up correctly, as shown in Figure 1-8. For example, if you have a sprite-displayed explosion, the first cells may be smaller than the last. By setting the registration point to the center of each image, the explosion animation will be centered at the sprite's location defined by its matrix.

Figure 1-8 Centered registration points

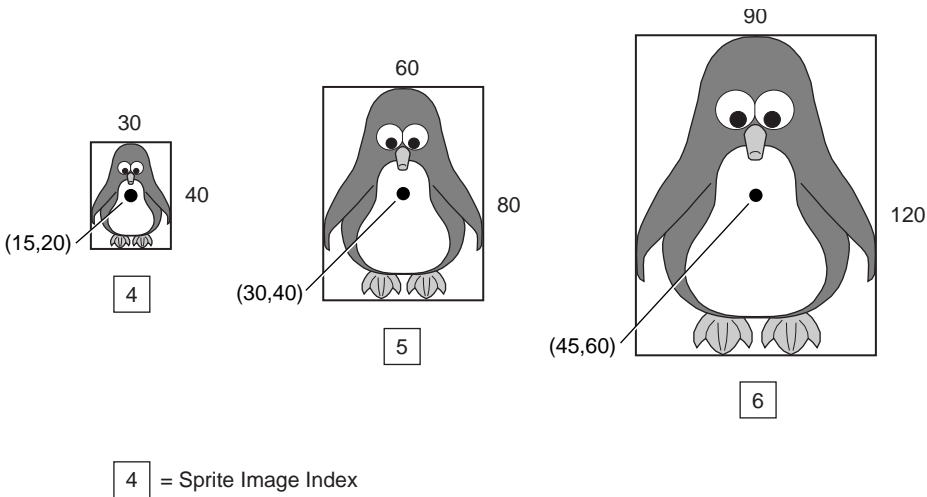
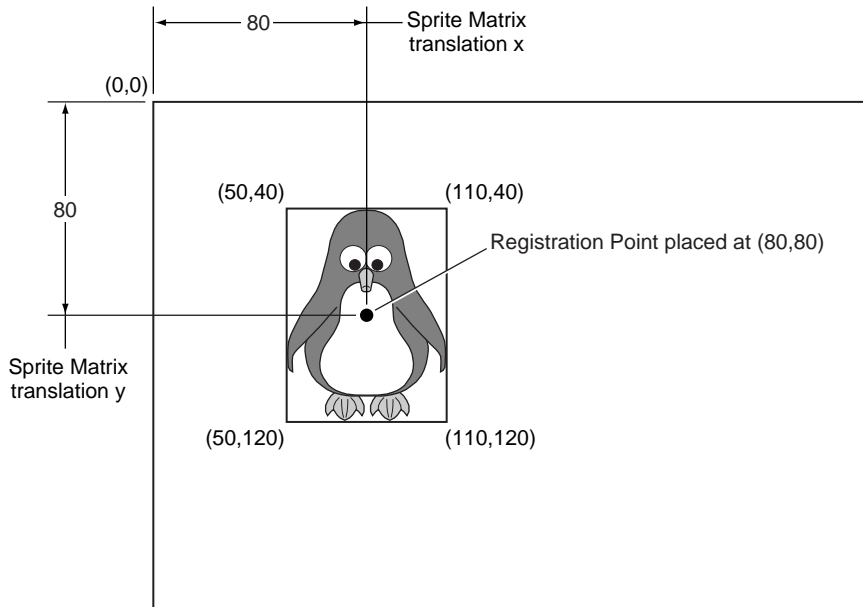


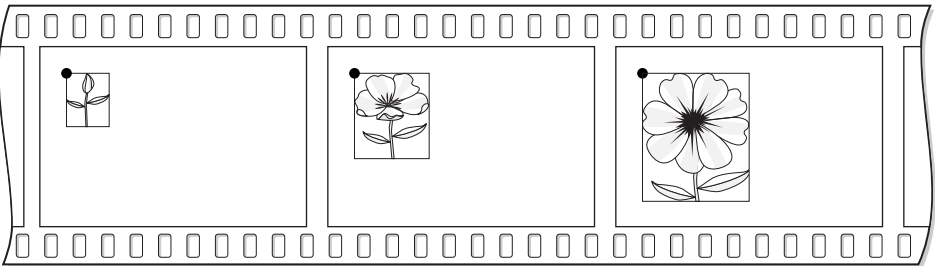
Figure 1-9 shows a centered registration point in a sprite track's local coordinate system.

Figure 1-9 A centered registration point in a sprite track's local coordinate system**Note**

When a sprite uses images of different sizes, you assign group IDs to the images. (For more information on group IDs, see “Assigning Group IDs” (page 40)). Group IDs are illustrated in the code sample in Appendix C. ♦

Figure 1-10 shows an example of registration points in a QuickTime movie.

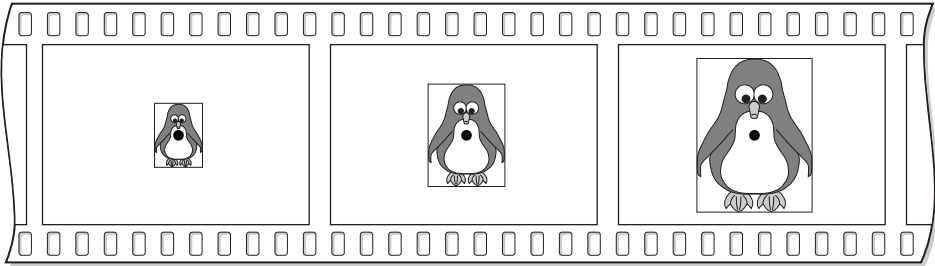
Figure 1-10 Registration points in a QuickTime movie



Sprite Matrix Translation x	20	20	20
Sprite Matrix Translation y	20	20	20
Sprite Image Index	1	2	3

Figure 1-11 shows an example of centered registration points in a QuickTime movie.

Figure 1-11 Centered registration points in a QuickTime movie



Sprite Matrix Translation x	100	100	100
Sprite Matrix Translation y	80	80	80
Sprite Image Index	4	5	6

Display Space

Display space, shown in Figure 1-12, refers to the pixels drawn in a window. In order to determine the area that a sprite is drawn to in display space, its registration point is first applied to its source matrix. This result is concatenated with its track's matrix and then concatenated with its movie's matrix.

Figure 1-12 A sprite display space and movie matrix identity

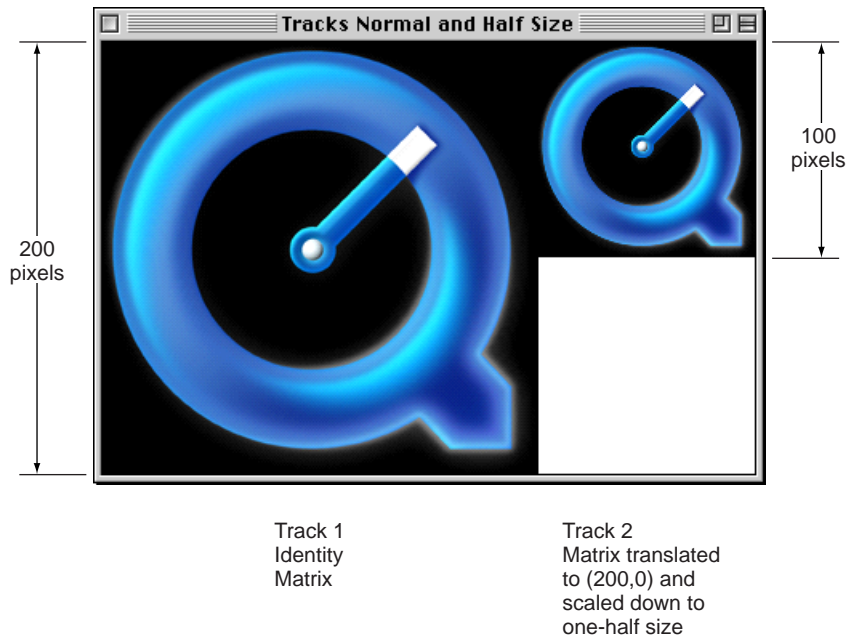
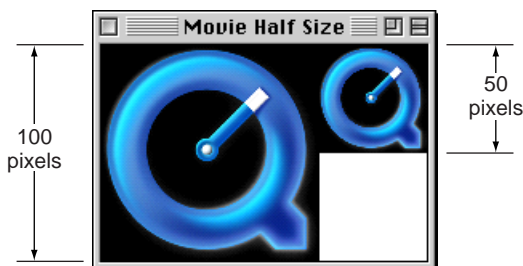


Figure 1-13 shows a movie matrix scaled down to one-half size.

Figure 1-13 A movie matrix scaled down to one-half size

Sprite Properties

A sprite's matrix property (`kSpritePropertyMatrix`) describes the sprite's location and scaling within its sprite world or sprite track. By modifying a sprite's matrix, you can modify the sprite's location so that it appears to move in a smooth path on the screen or so that it jumps from one place to another. You can modify a sprite's size, so that it shrinks, grows, or stretches. Depending on which image compressor is used to create the sprite images, other transformations, such as rotation, may be supported as well. Translation-only matrices provide the best performance.

A sprite's layer property (`kSpritePropertyLayer`) is a numeric value that specifies a sprite's layer in the animation. Sprites with lower layer numbers appear in front of sprites with higher layer numbers. To designate a sprite as a background sprite, you should assign it the special layer number `kBackgroundSpriteLayerNum`.

A sprite's visible property (`kSpritePropertyVisible`) specifies whether or not the sprite is visible. To make a sprite visible, you set the sprite's visible property to `TRUE`.

A sprite's graphics mode property (`kSpritePropertyGraphicsMode`) specifies a graphics mode and blend color that indicates how to blend a sprite with any sprites behind it and with the background. To set a sprite's graphics mode, you call `SetSpriteProperty`, passing a pointer to a `ModifierTrackGraphicsModeRecord` structure.

Defining QuickTime Atoms and Atom Containers

QuickTime stores most of its data using specialized structures in memory, called **atoms**. Movies themselves are atoms, as are tracks, media, and data samples. There are two kinds of atoms: **chunk atoms**, which your code accesses by offsets, and **QT atoms**, for which QuickTime provides a full set of access tools.

Each atom carries its own size and type information as well as its data. A **container atom** is an atom that contains other atoms, including other container atoms. There are several advantages to using QT atoms for holding and passing information:

QT atoms can nest indefinitely, forming hierarchies that are easy to pass from one process to another.

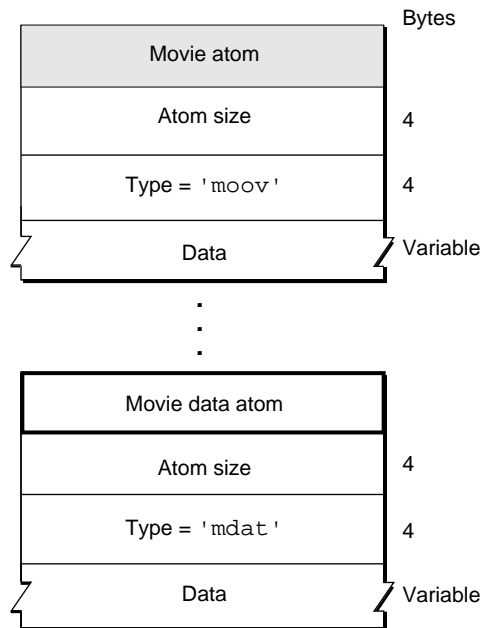
QuickTime provides a single set of tools by which you can search and manipulate QT atoms of all types.

Each atom has a four-character type designation that describes its internal structure. For example, movie atoms are type 'moov', while the track atoms inside them are type 'trak'.

Atoms that contain only data, and not other atoms, are called **leaf atoms**. A leaf atom simply contains a series of data fields accessible by offsets. You can use QuickTime's atom tools to search through QT atom hierarchies until you get to leaf atoms, then read the leaf atom's data from its various fields. With chunk atoms, you read their size bytes and access their contents by calculating offsets. For more information about atoms and atom containers, see *Inside Macintosh: QuickTime*.

Figure 1-14 shows an example of the atom structure of a simple QuickTime movie that has one track containing video data. Both the atoms in Figure 1-14 are chunk atoms, so you create and read them through your own code.

Figure 1-14 Atom structure of a simple QuickTime movie



The Sprite Media Handler

This chapter describes the **sprite media handler**, a media handler you can use to add a sprite animation track to a QuickTime movie. The sprite media handler provides routines for manipulating the sprites and images in a sprite track. The sprite media handler makes use of the functionality provided by the sprite toolbox discussed in Chapter 1, “Introduction to Wired Movies, Sprites, and the Sprite Toolbox.” If you are using the sprite media handler, you don’t need to use the toolbox API.

This chapter is divided into the following major sections:

- “About the Sprite Media Handler” (page 35) discusses the sprite media handler, which provides routines for manipulating the sprites and images in a sprite track.
- “New Features in QuickTime 4” (page 46) describes the new features available to your application in QuickTime 4, including new sprite button behaviors and variable string support.

For more information about the constants and functions in the sprite media handler, refer to Chapter 8, “Sprite Media Handler API Reference.”

About the Sprite Media Handler

The sprite media handler is a media handler that makes it possible to add a track containing a sprite animation to a QuickTime movie. The sprite media handler provides routines for manipulating the sprites and images in a sprite track.

The sprite media handler makes use of routines provided by the sprite toolbox. For background information about sprites, sprite animation, and the sprite

toolbox, see Chapter 1, “Introduction to Wired Movies, Sprites, and the Sprite Toolbox.”

As with sprites created in a sprite world, sprites in a sprite track have properties that define their locations, images, and appearance. However, you create the sprite track and its sprites differently than you create the sprites in a sprite world.

A **sprite track** is defined by one or more key frame samples, each followed by any number of override samples. A key frame sample and its subsequent override samples define a scene in the sprite track. A key frame sample is a QT atom container that contains atoms defining the sprites in the scene and their initial properties. The override samples are other QT atom containers that contain atoms that modify sprite properties, thereby animating the sprites in the scene. A sprite track sample is a QT atom container structure. In addition to defining properties for individual sprites, you can also define properties that apply to an entire sprite track.

For more information about QT atoms and atom containers, see Chapter 1, “Introduction to Wired Movies, Sprites, and the Sprite Toolbox.”

A key frame sample also contains all of the images used by the sprites. This allows the sprites in a sprite track to share image data. The images consist of two parts, an image description handle (`ImageDescriptionHandle`) concatenated with compressed image data. The image description handle describes the compressed image. You can compress the image using any QuickTime codec.

Images are stored in a key frame sample by index; each sprite has an image index property (`kSpritePropertyImageIndex`) that specifies the sprite’s current image. All images assigned to a sprite must be created using the same image description, unless you use group IDs.

The matrix, layer, visible, and graphics mode sprite properties have the same meaning for a sprite in a sprite track as for a sprite created in a sprite world.

As with sprite worlds, you can create a sprite track that has a solid background color, a background image composed of the images of one or more background sprites, or both a background color and a background image.

Hit Testing Flags

The following hit testing flags were introduced in QuickTime 3. These flags are for use with `SpriteMediaHitTestAllSprites` and `SpriteMediaHitTestOneSprite`:

The Sprite Media Handler

- `spriteHitTestInvisibleSprites`, which you set if you want invisible sprites to be hit tested along with visible ones.
- `spriteHitTestLocInDisplayCoordinates`, which you set if the hit testing point is in display coordinates instead of local sprite track coordinates.
- `spriteHitTestIsClick`, which you set if you want the hit testing operation to pass a click on to the codec currently rendering the sprites image. For example, this can be used to make the Ripple codec ripple.

Key Frame Samples and Override Samples

As discussed, a sprite track is defined by one or more key frame samples, each followed by any number of override samples. A key frame sample for a sprite track defines the following aspects of a sprite track:

- The number of sprites in the scene and their initial properties.
- All of the shared image data to be used by the sprites in the scene, including image data to be used in the subsequent override samples. Because a key frame sample contains the image data for the scene, the key frame sample tends to be larger than its subsequent override samples.

An override sample overrides some aspect of the key frame sample. For example, an override sample might modify the location of sprites defined in the key frame sample. Override samples do not contain any image data, so they can be very small. An override sample can show or hide a sprite defined in the key frame sample, but it cannot define new sprites or remove sprites defined in its key frame sample. An override sample can override any number of properties for any number of sprites. For example, a single override sample might change the layer and location of sprite ID 3, and hide sprite ID 10.

There are two sprite track formats that define how a key frame sample and its subsequent override samples are interpreted. If the current sample is a key frame sample, the key frame sample alone fully describes the current state of the track. If the current sample is an override sample, the current state may differ depending on the sprite track format:

- If the sprite track format is `kKeyFrameAndSingleOverride`, the current state is defined by the most recent key frame sample and the current override sample. This is the default format. The advantage of this format is that it allows for excellent performance during random access. A sprite track that uses this format can play backwards and drop frames smoothly. The

disadvantage of this format is that the file size of the track may be larger than a track that uses the other format.

- If the sprite track format is `kKeyFrameAndAllOverrides`, the current state is defined by the most recent key sample and all subsequent override samples, including the current override sample. This format results in a smaller file size. However, you should not use this format if you want your sprite track to play backwards or drop frames smoothly. When you play a movie that contains a sprite track whose format is `kKeyFrameAndAllOverrides`, you should configure the movie to play all frames.

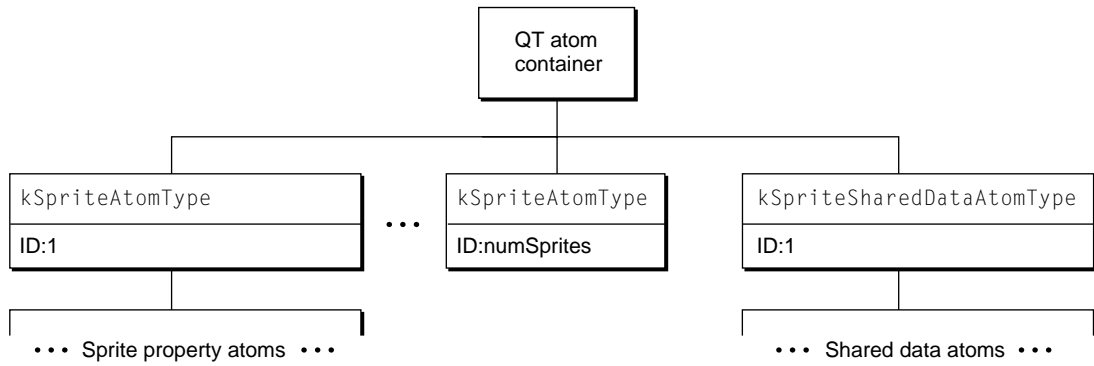
IMPORTANT

A sprite track must be authored exclusively with a single format, i.e., either all with `kKeyFrameAndSingleOverride` or all with `kKeyFrameAndAllOverrides`. ▲

Sprite Track Media Format

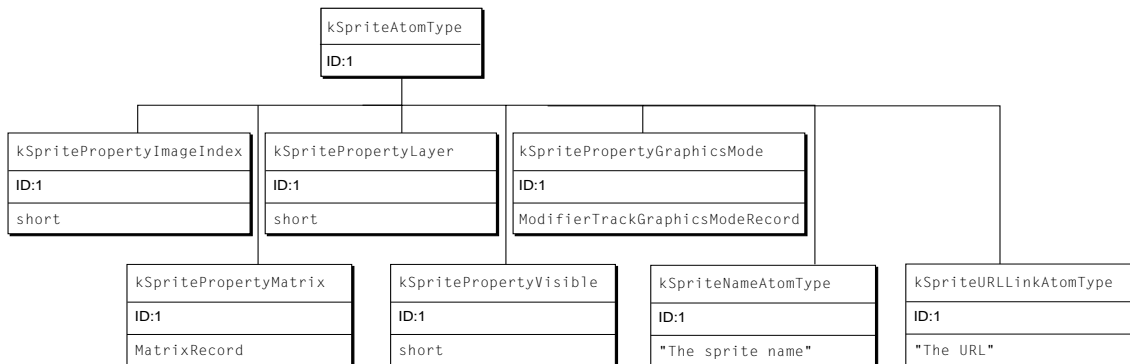
The sprite track media format is hierarchical and is based on QT atoms and atom containers. A sprite track sample is a QT atom container structure. For more information on QT atoms and atom containers, see Chapter 1, “Introduction to Wired Movies, Sprites, and the Sprite Toolbox.” For more information about the sprite media format, see Appendix B in this book. For information about a key to `QTAtomContainer`-based data structures that are being widely used in QuickTime, see Appendix A.

Figure 2-1 shows the high-level structure of a sprite track key frame sample. A key frame sample is represented by a QT atom container. Each atom in the atom container is represented by its atom type, atom ID, and, if it is a leaf atom, the type of its data.

Figure 2-1 A key frame sample atom container

The QT atom container contains one child atom for each sprite in the key frame sample. Each sprite atom has a type of `kSpriteAtomType`. The sprite IDs are numbered from one to the number of sprites defined by the key frame sample (`numSprites`).

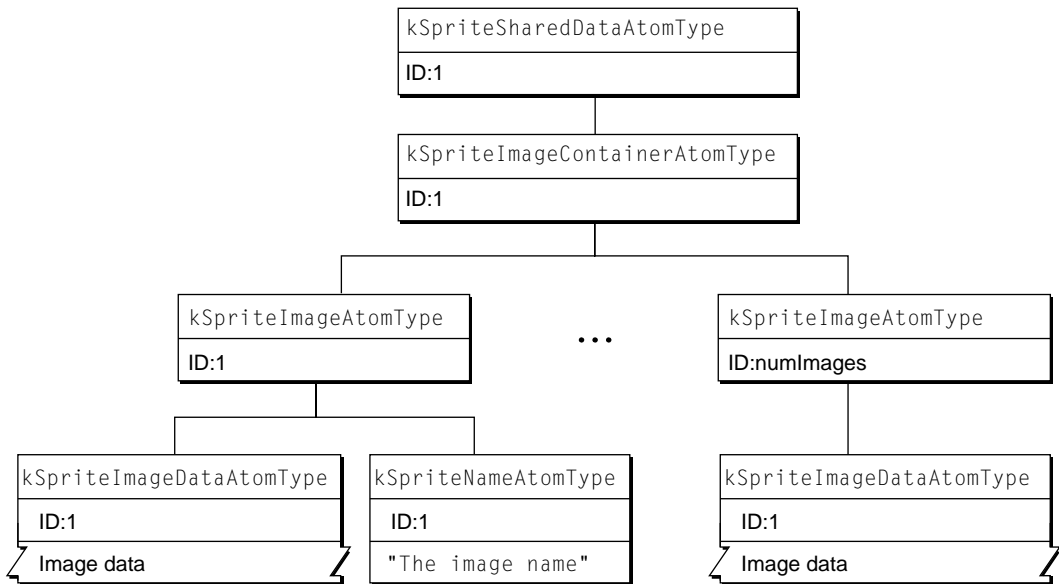
Each sprite atom contains leaf atoms that define the properties of the sprite, as shown in Figure 2-2. For example, the `kSpritePropertyLayer` property defines a sprite's layer. Each sprite property atom has an atom type that corresponds to the property and an ID of 1.

Figure 2-2 Atoms that describe a sprite and its properties

In addition to the sprite atoms, the QT atom container contains one atom of type `kSpriteSharedDataAtomType` with an ID of 1. The atoms contained by the shared data atom describe data that is shared by all sprites. The shared data atom contains one atom of type `kSpriteImagesContainerAtomType` with an ID of 1 (Figure 2-3).

The image container atom contains one atom of type `kImageAtomType` for each image in the key frame sample. The image atom IDs are numbered from one to the number of images (`numImages`). Each image atom contains a leaf atom that holds the image data (type `kSpriteImageDataAtomType`) and an optional leaf atom (type `kSpriteNameAtomType`) that holds the name of the image.

Figure 2-3 Atoms that describe sprite images



Assigning Group IDs

Before QuickTime 3, sprites could only display images with the same image description. This restriction has been relaxed, but you must assign group IDs to sets of equivalent images in your key frame sample. For example, if the sample contains 10 images where the first 2 images are equivalent, and the last 8 images

The Sprite Media Handler

are equivalent, you could assign a group ID of 1000 to the first 2 images, and a group ID of 1001 to the last 8 images. This divides the images in the sample into two sets. The actual ID does not matter; it just needs to be a unique positive integer.

Each image in a sprite media key frame sample is assigned to a group. You add an atom of type `kSpriteImageGroupIDAtomType` as a child of the `kSpriteImageAtomType` atom and set its leaf data to a `long` containing the group ID.

You must assign group IDs to your sprite sample if you want a sprite to display images with non-equivalent image descriptions (i.e., images with different dimensions).

Sprite Image Registration

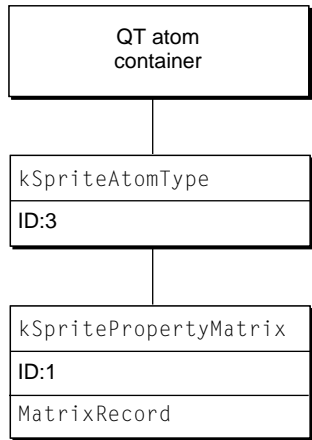
Sprite images have a default registration point of 0, 0. To specify a different point, you add an atom of type `kSpriteImageRegistrationAtomType` as a child atom of the `kSpriteImageAtomType` and set its leaf data to a `FixedPoint` value with the desired registration point.

The format of an override sample is identical to that of a key frame sample with the following exceptions:

- An override sample does not contain images, which means it does not contain an atom of type `kSpriteImagesContainerAtomType` or any of its children.
- In an override sample, all of the sprite atoms and sprite property atoms are optional.

For example, to define an override sample that modifies the location of the third sprite defined by the previous key frame sample, you would create a QT atom container and add the following atoms to it (assuming that the sprite track format is of type `kKeyFrameAndSingleOverride`):

Figure 2-4 An example of an override sample atom container



Sprite Track Properties

In addition to defining properties for individual sprites, you can also define properties that apply to an entire sprite track. These properties may override default behavior or provide hints to the sprite media handler. The following sprite track properties are supported:

- The `kSpriteTrackPropertyBackgroundColor` property specifies a background color for the sprite track. The background color is used for any area that is not covered by regular sprites or background sprites. If you do not specify a background color, the sprite track uses black as the default background color.
- The `kSpriteTrackPropertyOffscreenBitDepth` property specifies a preferred bit depth for the sprite track's offscreen buffer. The allowable values are 8 and 16. To save memory, you should set the value of this property to the minimum depth needed. If you do not specify a bit depth, the sprite track allocates an offscreen buffer with the depth of the deepest intersecting monitor.
- The `kSpriteTrackPropertySampleFormat` property specifies the sample format for the sprite track. If you do not specify a sample format, the sprite track uses the default format, `kKeyFrameAndSingleOverride`.

The Sprite Media Handler

For wired sprites, which are discussed in Chapter 1, “Introduction to Wired Movies, Sprites, and the Sprite Toolbox,” the following sprite track properties are supported:

- The `SpriteTrackPropertyHasActions` property. You must add an atom of this type with its leaf data set to `true` if you want the movie controller to execute the actions in your sprite track’s media. The atom’s leaf data is of type `Boolean`. The default value is `false`, so it is very important to add an atom of this type if you want interactivity to take place.
- The `kSpriteTrackPropertyQTIdleEventsFrequency` property. You must add an atom of this type if you want the sprites in your sprite track to receive `kQTEventIdle` `QTEvents`. The atom’s leaf data is of type `UInt32`. The value is the minimum number of ticks that must pass before the next `QTIdle` event is sent. Each tick is 1/60th of one second. For more information, see “Sprite Track Property Atoms” (page 162).
- The `kSpriteTrackPropertyVisible` property. You may cause the entire sprite track to be invisible by setting the value of this Boolean property to `false`. This is useful for using a sprite track as a hidden button track—for example, placing an invisible sprite track over a video track would allow the characters in the video to be clicked on. The default value is `visible` (`true`).
- The `kSpriteTrackPropertyScaleSpritesToScaleWorld` property. You may cause each sprite to be rescaled when the sprite track is resized by setting the value of this Boolean property to `true`. Setting this property can improve the drawing performance of a scaled sprite track. This is particularly useful for sprite images compressed with codecs which are resolution-independent, such as the Curve codec. The default value for this property is `false`.

To specify sprite track properties, you create a single QT atom container and add a leaf atom for each property you want to specify. To add the properties to a sprite track, you call the new media handler function `SetMediaPropertyAtom`. To retrieve a sprite track’s properties, you call the media handler function `GetMediaPropertyAtom`.

The sprite track properties and their corresponding atom data are outlined in Table 2-1.

Table 2-1 Sprite track properties

Atom type	Atom ID	Leaf data type
kSpriteTrackPropertyBackgroundColor	1	RGBColor
kSpriteTrackPropertyOffscreenBitDepth	1	unsigned short
kSpriteTrackPropertySampleFormat	1	long
kSpriteTrackPropertyHasActions	1	Boolean
kSpriteTrackPropertyQTIdleEventsFrequency	1	UInt32
kSpriteTrackPropertyVisible	1	Boolean
kSpriteTrackPropertyScaleSpritesToScaleWorld	1	Boolean

Note

When pasting portions of two different tracks together, the Movie Toolbox checks to see that all sprite track properties match. If, in fact, they do match, the paste results in a single sprite track instead of two. ♦

Alternate Sources for Sprite Image Data

A sprite in a sprite track can obtain its image data from sources other than the images in the sprite track’s key frame sample. The alternate image data overrides a particular image index in the sprite track so that all sprites with that image index will use the image data provided by the alternate source.

A sprite track can receive image data from another track within the same movie, called a modifier track. This is useful for compositing traditional video tracks with sprites. For example, you might create a sprite track in which sprite characters are watching television. The sprite track can receive video from another track, called a modifier track, to use as the image data for the television screen sprite. Other sprites can move in front of and behind the television. A

sprite track can have more than one modifier track feeding it image data and more than one sprite can use the image data from a modifier track at one time.

In order for a sprite to receive image data from a modifier track, you must call the `AddTrackReference` function to link the modifier track to the sprite track that it modifies. In addition, you must update the sprite media's input map with an atom that specifies the input type (`kTrackModifierTypeImage`) and an atom that specifies the index of the image to replace (`kSpritePropertyImageIndex`).

A sprite track can also receive sprite image data from an application. For example, an application might provide live, digitized video data to a sprite track by calling `MediaSetNonPrimarySourceData`.

Supported Modifier Inputs

In addition to receiving image data, a sprite track can receive modifier track data to control its sprites. The following modifier inputs are supported:

- images from a video track (`kTrackModifierTypeImage`)
- a matrix from a base track (`kTrackModifierObjectMatrix`)
- a graphics mode from a base track (`kTrackModifierObjectGraphicsMode`)
- an image index from a base track (`kTrackModifierObjectImageIndex`)
- an object layer from a base track (`kTrackModifierObjectLayer`)
- an object visible from a base track (`kTrackModifierObjectVisible`)

For example, a modifier track can send matrices to individual sprites to control their locations. To do this, you set up a modifier track, such as a tween track, to send matrix data to the sprite track. You must update the sprite media's input map with an atom that specifies the input type (`kTrackModifierObjectMatrix`) and an atom that specifies the ID of the sprite to replace (`kTrackModifierObjectID`). If the sprite track also contains matrices to move the sprites, the results are undefined.

Note

With the exception of image data, the source for all modifier tracks can be tween or base tracks. ♦

For background information on modifier tracks, see Chapter 1, “Movie Toolbox,” in *QuickTime 4 Reference*.

New Features in QuickTime 4

This section describes the new features available to your application in QuickTime 4.

Referenced Sprite Images

In QuickTime 4 the sprite track can use images which are external to its movie's media. The images may be located somewhere on the Internet, in a local file, or anywhere else that a QuickTime Data Handler can read them from.

Since images located on a network server may take some time to load (or possibly never show up), referenced images are loaded asynchronously. Sprites *not* using referenced images will be created and will be active while the referenced images are loading. You may optionally supply a proxy image that will be displayed until the referenced image has been loaded. If you don't supply a proxy image, sprites using the referenced image will be disabled (invisible and not responsive to mouse events) until it is loaded.

Otherwise, referenced images are the same as traditional ones. They are defined in a sprite key frame sample, available until the next key frame sample is loaded, used by a sprite setting its `imageIndex` property to the images index, and may be shared by multiple sprites.

Sprite Button Behaviors

In QuickTime 4 sprites in a sprite track may specify some simple button behaviors. These behaviors may control the sprite's image, the system cursor, and the status message displayed in a Web browser. These behaviors are a compact shortcut for a very common set of actions which result in more efficient movies.

Button behaviors may also be added to a sprite. These behaviors are intended to make the common task of creating buttons in a sprite track easy—you basically just fill in a template. Three types of behaviors are available; you choose one or more of them. The behaviors each change a type of property associated with a button and are triggered by the mouse states `notOverNotPressed`,

The Sprite Media Handler

overNotPressed, overPressed, and notOverPressed. The three properties changed are

- the sprites' `imageIndex`
- the ID of a cursor to be displayed
- the ID of a status string variable displayed in the URL status area of a Web browser.

Setting a property's value to -1 means don't change it.

Note

The cursor is currently automatically set back to the default system cursor when leaving a sprite. ♦

The sprite track handles letting one sprite act as an active button at a time.

The behaviors are prepended to the sprite's list of actions, so they may be overridden by actions if desired.

To use the behaviors, you fill in the new atoms as follows:

```
kSpriteAtomType
    <kSpriteBehaviorsAtomType>, 1
        <kSpriteImageBehaviorAtomType>
            [QTSpriteButtonBehaviorStruct]
        <kSpriteCursorBehaviorAtomType>
            [QTSpriteButtonBehaviorStruct]
        <kSpriteStatusStringsBehaviorAtomType>
            [QTSpriteButtonBehaviorStruct]
```

Action Handler Property

QuickTime 4 adds a new action handler property:

`kSpritePropertyActionHandlingSpriteID` whose data type is `QTAtomID`. You set this new sprite property to the ID of another sprite in the sprite track that you wish to delegate QT event handling to.

String Variable Support

In QuickTime 4 sprite track variables may contain either strings or floating point numbers. These variables may be used for all action and operand

parameters that accept strings, such as `GotoURL`. Additionally, they may be concatenated together to create new string variables.

The new `kActionSpriteTrackSetVariableToString` action has been introduced to allow you to set a variable to a string value. You still use the `kActionSpriteTrackSetVariable` action to set a variable to a floating-point number.

When a sprite track variable is retrieved and used via the `kOperandSpriteTrackVariable` operand, it will be coerced to the required type. If it is used as part of an expression and it is a string, it will be converted to a floating-point number. If used as a string parameter, it will be converted to a C or Pascal string as needed from a floating-point number or string variable.

When a floating point number which does not contain an integer value is coerced to a string, a format of up to five digits before the decimal point and three digits afterwards is used.

Authoring Wired Movies and Sprite Animations

This chapter describes how you can author wired movies and sprite animations using the sprite media handler. You use the functions provided by the sprite media handler to create and manipulate a sprite animation as a track in a QuickTime movie. You can also use the functions provided by the sprite media handler to create and manipulate a wired sprite movie, with various types of user interactivity.

The chapter is illustrated with code snippets from the sample program, `QTWiredSprites.c`, which is listed in full in Appendix C.

This chapter also discusses briefly some of the new authoring features provided in QuickTime 4.

The chapter is divided into the following major sections:

- “Authoring Movies with the Sprite Media Handler” (page 49)
- “Authoring Wired Movies” (page 65)
- “New Authoring Features in QuickTime 4” (page 73)
- “Wiring a Movie by Adding an HREF Track” (page 76)

For more conceptual information about the sprite media handler, refer to Chapter 2, “The Sprite Media Handler.” For API reference information about the constants and functions available to your application, refer to Chapter 8, “Sprite Media Handler API Reference.”

Authoring Movies with the Sprite Media Handler

The sprite media handler provides functions that allow an application to create and manipulate a sprite animation as a track in a QuickTime movie.

The following sections are illustrated with code from the sample program `QTWiredSprites.c`, which creates a 320 by 240 pixel QuickTime movie with one sprite track. The sprite track contains six sprites, including two penguins and four buttons. The sample program, which takes advantage of wired sprites, is explained in greater detail “Authoring Wired Movies” (page 65). A partial code listing is available in Appendix C. You can download the full sample code at QuickTime Web site at <http://www.apple.com/quicktime/developers/samplecode.html#sprites>.

Defining a Key Frame Sample

To create a sprite track in a QuickTime movie, you must first create the movie itself, a track to contain the sprites, and the track’s media.

After doing this, you can define a key frame sample. A **key frame sample** defines the number of sprites, their initial property values, and the shared image data used by the sprites in the key frame sample and in all override samples that follow the key frame sample. The sample code discussed in this section creates a single key frame sample and shows how to add images for other sprites, as well as actions for other sprites.

Creating the Movie, Sprite Track, and Media

Listing 3-1 shows a code fragment from the sample code `QTWiredSprites.c`. This sample code, which is available in Appendix C, illustrates how you can create a new movie file that calls a sample code function, `AddSpriteTrackToMovie`, which is responsible for creating a sprite track and adding it to the movie.

Listing 3-1 Creating a sprite track movie

```
// Create a QuickTime movie containing a wired sprites track
.
.
.

// create a new movie file and set its controller type
// ask the user for the name of the new movie file
StandardPutFile("\pSprite movie file name:", "\pSprite.mov",
               &myReply);
if (!myReply.sfGood)
    goto bail;
```

Authoring Wired Movies and Sprite Animations

```

// create a movie file for the destination movie
myErr = CreateMovieFile(&myReply.sfFile, FOUR_CHAR_CODE('TVOD'), 0,
                      myFlags, &myResRefNum, &myMovie);
if (myErr != noErr)
    goto bail;

// select the "no controller" movie controller
myType = EndianU32_NtoB(myType);
SetUserDataItem(GetMovieUserData(myMovie), &myType, sizeof(myType),
                kUserDataMovieControllerType, 1);

```

The following code fragment from `AddSpriteTrackToMovie` (Listing 3-2) creates a new track and new media, and creates an empty key frame sample.

`AddSpriteTrackToMovie` then calls `BeginMediaEdits` (Listing 3-4) to prepare to add samples to the track's media.

Listing 3-2 Creating a track and media

```

// create the sprite track and media

myTrack = NewMovieTrack(myMovie, ((long)kSpriteTrackWidth << 16),
                        ((long)kSpriteTrackHeight << 16), kNoVolume);
myMedia = NewTrackMedia(myTrack, SpriteMediaType,
                        kSpriteMediaTimeScale, NULL, 0);

// create a new, empty key frame sample
myErr = QTNewAtomContainer(&mySample);
if (myErr != noErr)
    goto bail;

myKeyColor.red = 0xffff; // white
myKeyColor.green = 0xffff;
myKeyColor.blue = 0xffff;

```

Adding Images to the Key Frame Sample

The `AddPictImageToKeyFrameSample` function (Listing 3-3) adds images to the key frame sample.

Listing 3-3 Adding images to the key frame sample

```
// add images to the key frame sample
    AddPictImageToKeyFrameSample(mySample, kGoToBeginningButtonUp,
                                &myKeyColor, kGoToBeginningButtonUpIndex, NULL, NULL);
    AddPictImageToKeyFrameSample(mySample, kGoToBeginningButtonDown,
                                &myKeyColor, kGoToBeginningButtonDownIndex, NULL, NULL);
    AddPictImageToKeyFrameSample(mySample, kGoToEndButtonUp, &myKeyColor,
                                kGoToEndButtonUpIndex, NULL, NULL);
    ...
    AddPictImageToKeyFrameSample(mySample, kPenguinForward, &myKeyColor,
                                kPenguinForwardIndex, NULL, NULL);
    AddPictImageToKeyFrameSample(mySample, kPenguinLeft, &myKeyColor,
                                kPenguinLeftIndex, NULL, NULL);
    AddPictImageToKeyFrameSample(mySample, kPenguinRight, &myKeyColor,
                                kPenguinRightIndex, NULL, NULL);
    AddPictImageToKeyFrameSample(mySample, kPenguinClosed, &myKeyColor,
                                kPenguinClosedIndex, NULL, NULL);

    for (myIndex = kPenguinDownRightCycleStartIndex, myResID =
         kWalkkDownRightCycleStart; myIndex <=
kPenguinDownRightCycleEndIndex;
         myIndex++, myResID++)
        AddPictImageToKeyFrameSample(mySample, myResID, &myKeyColor, myIndex,
                                     NULL, NULL);
```

Adding More Images for Other Sprites

To add more images to other sprites, you assign group IDs to those images, using the `AssignImageGroupIDsToKeyFrame` function. You then create the sprite track, add it to the movie, and then begin to add samples to the tracks' media, as shown in Listing 3-4.

Listing 3-4 Adding more images to other sprites and specifying button actions

```
// assign group IDs to the images
    AssignImageGroupIDsToKeyFrame(mySample);

    // add samples to the sprite track's media
```

```
//

BeginMediaEdits(myMedia);

// go to beginning button with no actions
myErr = QTNewAtomContainer(&myBeginButton);
if (myErr != noErr)
    goto bail;
myLocation.h    = (1 * kSpriteTrackWidth / 8) - (kStartEndButtonWidth
                                                    / 2);
myLocation.v    = (4 * kSpriteTrackHeight / 5) -
                  (kStartEndButtonHeight / 2);

isVisible       = false;
myLayer         = 1;
myIndex         = kGoToBeginningButtonUpIndex;
myErr = SetSpriteData(myBeginButton, &myLocation, &isVisible,
                      &myLayer, &myIndex, NULL, NULL, myActions);

if (myErr != noErr)
    goto bail;
```

Adding Sprites to the Key Frame Sample

The `AddSpriteTrackToMovie` function adds the sprites with their initial property values to the key frame sample, as shown in Listing 3-5. The key frame contains four buttons and two penguins.

If the `withBackgroundPicture` parameter is `true`, the function adds a background sprite. The function initializes the background sprite's properties, including setting the layer property to `kBackgroundSpriteLayerNum` to indicate that the sprite is a background sprite. The function calls `SetSpriteData` (Listing 3-6), which adds the appropriate property atoms to the `spriteData` atom container. Then, `AddSpriteTrackToMovie` calls `AddSpriteToSample` (Listing 3-7) to add the atoms in the `spriteData` atom container to the key frame sample atom container.

`AddSpriteTrackToMovie` adds the other sprites to the key frame sample and then calls `AddSpriteSampleToMedia` (Listing 3-8) to add the key frame sample to the media.

Listing 3-5 Creating more key frame sprite media

```

// add actions to the six sprites
//
// add go to beginning button
myErr = QTCopyAtom(myBeginButton, kParentAtomIsContainer,
                  &myBeginActionButton);
if (myErr != noErr)
    goto bail;

AddSpriteSetImageIndexAction(myBeginActionButton,
    kParentAtomIsContainer, kQTEventMouseClicked, 0, NULL, 0, 0, NULL,
    kGoToBeginningButtonDownIndex, NULL);
AddSpriteSetImageIndexAction(myBeginActionButton,
    kParentAtomIsContainer, kQTEventMouseClickedEnd, 0, NULL, 0, 0,
    NULL, kGoToBeginningButtonUpIndex, NULL);
AddMovieGoToBeginningAction(myBeginActionButton,
    kParentAtomIsContainer, kQTEventMouseClickedEndTriggerButton);
AddSpriteSetVisibleAction(myBeginActionButton,
    kParentAtomIsContainer, kQTEventMouseEnter, 0, NULL, 0, 0, NULL,
    true, NULL);
AddSpriteSetVisibleAction(myBeginActionButton,
    kParentAtomIsContainer, kQTEventMouseExit, 0, NULL, 0, 0, NULL,
    false, NULL);
AddSpriteToSample(mySample, myBeginActionButton,
    kGoToBeginningSpriteID);
QTDisposeAtomContainer(myBeginActionButton);

// add go to prev button
myErr = QTCopyAtom(myPrevButton, kParentAtomIsContainer,
                  &myPrevActionButton);
if (myErr != noErr)
    goto bail;

AddSpriteSetImageIndexAction(myPrevActionButton,
    kParentAtomIsContainer, kQTEventMouseClicked, 0, NULL, 0, 0, NULL,
    kGoToPrevButtonDownIndex, NULL);
AddSpriteSetImageIndexAction(myPrevActionButton,
    kParentAtomIsContainer, kQTEventMouseClickedEnd, 0, NULL, 0, 0,
    NULL, kGoToPrevButtonUpIndex, NULL);
AddMovieStepBackwardAction(myPrevActionButton,

```

```

        kParentAtomIsContainer, kQTEventMouseClickedEndTriggerButton);
    AddSpriteSetVisibleAction(myBeginActionButton,
        kParentAtomIsContainer, kQTEventMouseEnter, 0, NULL, 0, 0, NULL,
        true, NULL);
    AddSpriteSetVisibleAction(myBeginActionButton,
        kParentAtomIsContainer, kQTEventMouseExit, 0, NULL, 0, 0, NULL,
        false, NULL);
    AddSpriteToSample(mySample, myPrevActionButton, kGoToPrevSpriteID);

    QTDisposeAtomContainer(myPrevActionButton);

    // add go to next button
    myErr = QTCopyAtom(myNextButton, kParentAtomIsContainer,
        &myNextActionButton);
    if (myErr != noErr)
        goto bail;

```

For each new property value that is passed into it as a parameter, the `SetSpriteData` function (Listing 3-6) calls `QTFindChildByIndex` to find the appropriate property atom. If the property atom already exists in the QT atom container, `SetSpriteData` calls `QTSetAtomData` to update the property's value. If the property atom does not exist in the container, `SetSpriteData` calls `QTInsertChild` to insert a new property atom.

Listing 3-6 The `SetSpriteData` function

```

OSErr SetSpriteData (QTAtomContainer sprite, Point *location,
    short *visible, short *layer, short *imageIndex)
{
    OSErr    err = noErr;
    QTAtom   propertyAtom;

    if (location) {
        MatrixRecordmatrix;

        // set up the value for the matrix property
        SetIdentityMatrix (&matrix);
        matrix.matrix[2][0] = ((long)location->h << 16);
        matrix.matrix[2][1] = ((long)location->v << 16);
    }
}

```

```

        // if no matrix atom is in the container, insert a new one
        if ((propertyAtom = QTFindChildByIndex (sprite, 0,
            kSpritePropertyMatrix, 1, nil)) == 0)
            FailOSErr (QTInsertChild (sprite, 0, kSpritePropertyMatrix,
                1, 0, sizeof(MatrixRecord), &matrix, nil))
        // otherwise, replace the atom's data else
        FailOSErr (QTSetAtomData (sprite, propertyAtom,
            sizeof(MatrixRecord), &matrix));
    }

    // ...
    // handle other properties in a similar fashion
    // ...

    return err;
}

```

The `AddSpriteToSample` function (Listing 3-7) checks to see whether a sprite has already been added to a sample. If not, the function calls `QTInsertChild` to create a new sprite atom in the atom container that represents the sample. Then, `AddSpriteToSample` calls `QTInsertChildren` to insert the atoms in the sprite atom container as children of the newly created atom in the sample container.

Listing 3-7 The `AddSpriteToSample` function

```

OSErr AddSpriteToSample (QTAtomContainer theSample,
    QTAtomContainer theSprite, short spriteID)
{
    OSErr err = noErr;
    QTAtom newSpriteAtom;

    FailIf (QTFindChildByID (theSample, 0, kSpriteAtomType, spriteID,
        nil), paramErr);

    FailOSErr (QTInsertChild (theSample, 0, kSpriteAtomType, spriteID,
        0, 0, nil, &newSpriteAtom)); // index of zero means append
    FailOSErr (QTInsertChildren (theSample, newSpriteAtom, theSprite));
}

```



```
bail:
    return err;
}
```

The `AddSpriteSampleToMedia` function, shown in Listing 3-8, calls `AddMediaSample` to add either a key frame sample or an override sample to the sprite media.

Listing 3-8 The `AddSpriteSampleToMedia` function

```
OSErr AddSpriteSampleToMedia (Media theMedia, QTAAtomContainer sample,
    TimeValue duration, Boolean isKeyFrame)
{
    OSErr err = noErr;
    SampleDescriptionHandle sampleDesc = nil;

    FailMemErr (sampleDesc = (SampleDescriptionHandle) NewHandleClear(
        sizeof(SampleDescription)));

    FailOSErr (AddMediaSample (theMedia, (Handle) sample, 0,
        GetHandleSize(sample), duration, sampleDesc, 1,
        isKeyFrame ? 0 : mediaSampleNotSync, nil));

    bail:
        if (sampleDesc)
            DisposeHandle ((Handle)sampleDesc);

    return err;
}
```

Adding More Actions to Other Sprites

To set the movie's looping mode to palindrome, you add an action that is triggered when the key frame is loaded, as shown in Listing 3-9. This action is triggered every time the key frame is reloaded.

Listing 3-9 Adding more actions to other sprites

```

loopingFlags = loopTimeBase | palindromeLoopTimeBase;
FailOSErr( AddMovieSetLoopingFlagsAction( sample,
                                           kParentAtomIsContainer,
                                           kQTEventFrameLoaded, loopingFlags ) )

```

Adding Sample Data in Compressed Form

To add the sample data in a compressed form, you use a QuickTime DataCodec to perform the compression, as shown in Listing 3-10. You replace the sample utility `AddSpriteSampleToMedia` call with a call to the sample utility `AddCompressedSpriteSampleToMedia`.

Listing 3-10 Adding the key frame sample in compressed form

```

/* AddSpriteSampleToMedia(myMedia, mySample, kSpriteMediaFrameDuration,
   true, NULL); */
AddCompressedSpriteSampleToMedia(myMedia, mySample,
                                 kSpriteMediaFrameDuration, true, zlibDataCompressorSubType,
                                 NULL);

```

Defining Override Samples

Once you have defined a key frame sample for the sprite track, you can add any number of override samples to modify sprite properties.

Listing 3-11 shows the portion of the `AddSpriteTrackToMovie` function that adds override samples to the sprite track to make the first penguin sprite appear to waddle and move across the screen. For each override sample, the function modifies the first penguin sprite's image index and location. The function calls `SetSpriteData` to update the appropriate property atoms in the sprite atom container. Then, the function calls `AddSpriteToSample` to add the sprite atom container to the sample atom container. After all of the modifications have been made to the override sample, the function calls `AddSpriteSampleToMedia` to add the override sample to the media.

After adding all of the override samples to the media, `AddSpriteTrackToMovie` calls `EndMediaEdits` to indicate that it is done adding samples to the media.

Then, `AddSpriteTrackToMovie` calls `InsertMediaIntoTrack` to insert the new media segment into the track.

Listing 3-11 Adding override samples to move penguin one and change its image index

```
// original penguin one location
myLocation.h = (3 * kSpriteTrackWidth / 8) - (kPenguinWidth / 2);
myLocation.v = (kSpriteTrackHeight / 4) - (kPenguinHeight / 2);

myDelta = (kSpriteTrackHeight / 2) / kNumOverrideSamples;
myIndex = kPenguinDownRightCycleStartIndex;

for (i = 1; i <= kNumOverrideSamples; i++) {
    QTRemoveChildren(mySample, kParentAtomIsContainer);
    QTNewAtomContainer(&myPenguinOneOverride);

    myLocation.h += myDelta;
    myLocation.v += myDelta;
    myIndex++;
    if (myIndex > kPenguinDownRightCycleEndIndex)
        myIndex = kPenguinDownRightCycleStartIndex;

    SetSpriteData(myPenguinOneOverride, &myLocation, NULL, NULL,
                  &myIndex, NULL, NULL, NULL);
    AddSpriteToSample(mySample, myPenguinOneOverride,
                     kPenguinOneSpriteID);
    AddSpriteSampleToMedia(myMedia, mySample,
                          kSpriteMediaFrameDuration, false, NULL);
    QTDisposeAtomContainer(myPenguinOneOverride);
}

EndMediaEdits(myMedia);

// add the media to the track
InsertMediaIntoTrack(myTrack, 0, 0, GetMediaDuration(myMedia),
                    fixed1);
```

Setting Properties of the Sprite Track

Besides adding key frame samples and override samples to the sprite track, you may want to set one or more global properties of the sprite track. For example, if you want to define a background color for your sprite track, you must set the sprite track's background color property. You do this by creating a leaf atom of type `kSpriteTrackPropertyBackgroundColor` whose data is the desired background color.

After adding the override samples, `AddSpriteTrackToMovie` adds a background color to the sprite track, as shown in Listing 3-12. The function calls `QTNewAtomContainer` to create a new atom container for sprite track properties. `AddSpriteTrackToMovie` adds a new atom of type `kSpriteTrackPropertyBackgroundColor` to the container and calls `SpriteMediaSetSpriteProperty` to set the sprite track's property.

After adding a background color, `AddSpriteTrackToMovie` notifies the movie controller that the sprite track has actions. If the `hasActions` parameter is `true`, this function calls `QTNewAtomContainer` to create a new atom container for sprite track properties. `AddSpriteTrackToMovie` adds a new atom of type `kSpriteTrackPropertyHasActions` to the container and calls `SpriteMediaSetSpriteProperty` to set the sprite track's property.

Finally, after specifying that the sprite track has actions, `AddSpriteTrackToMovie` notifies the sprite track to generate `QTIdleEvents` by adding a new atom of type `kSpriteTrackPropertyQTIdleEventsFrequency` to the container. This new atom specifies the frequency of `QTEvent` occurrences.

Listing 3-12 Adding sprite track properties, including a background color, actions, and frequency

```
{
    QTAtomContainer    myTrackProperties;
    RGBColor           myBackgroundColor;

    // add a background color to the sprite track
    myBackgroundColor.red = EndianU16_NtoB(0x8000);
    myBackgroundColor.green = EndianU16_NtoB(0);
    myBackgroundColor.blue = EndianU16_NtoB(0xffff);

    QTNewAtomContainer(&myTrackProperties);
    QTInsertChild(myTrackProperties, 0,
```

```

        kSpriteTrackPropertyBackgroundColor, 1, 1,
        sizeof(RGBColor), &myBackgroundColor, NULL);

    // tell the movie controller that this sprite track has actions
    hasActions = true;
    QTInsertChild(myTrackProperties, 0,
        kSpriteTrackPropertyHasActions, 1, 1,
        sizeof(hasActions), &hasActions, NULL);

    // tell the sprite track to generate QTIdleEvents
    myFrequency = EndianU32_NtoB(60);
    QTInsertChild(myTrackProperties, 0,
        kSpriteTrackPropertyQTIdleEventsFrequency, 1, 1,
        sizeof(myFrequency), &myFrequency, NULL);
    myErr = SetMediaPropertyAtom(myMedia, myTrackProperties);
    if (myErr != noErr)
        goto bail;

    QTDisposeAtomContainer(myTrackProperties);
}

```

Getting Sprite Data From a Modifier Track

The sample program `AddReferenceTrack.c` illustrates how you can modify a movie to use a modifier track for a sprite's image data. The sample program prompts the user for a movie that contains a single sprite track. Then, it adds a track from a second movie to the original movie as a modifier track. The modifier track overrides the image data for a selected image index.

Listing 3-13 shows the first part of the main function of the sample program. It performs the following tasks:

- It loads the movie containing the sprite track.
- It calls `GetMovieTrackCount` to determine the total number of tracks in the sprite track movie.
- It loads the movie containing the modifier track (`movieB`).

Listing 3-13 Loading the movies

```

OSErr                err;
short                movieResID = 0, resFref, resID = 0, resRefNum;
StandardFileReply    reply;
SFTYPEList           types;
Movie                m;
FSSpec               fss;
Movie                movieB;
long                 origTrackCount;

// prompt for a movie containing a sprite track and load it
types[0] = MovieFileType;
StandardGetFilePreview (nil, 1, types, &reply);
if (!reply.sfGood) return;

err = OpenMovieFile (&reply.sfFile, &resFref, fsRdPerm);
if (err) return;

err = NewMovieFromFile (&m, resFref, &movieResID, (StringPtr)nil,
    newMovieActive, ni);
if (err) return;

CloseMovieFile (resFref);

// get the number of tracks
origTrackCount = GetMovieTrackCount (m);

// load the movie to be used as a modifier track
FSMakeFSSpec (reply.sfFile.vRefNum, reply.sfFile.parID, "\pAdd Me",
    &fss);

err = OpenMovieFile (&fss, &resFref, fsRdPerm);
if (err) return;

err = NewMovieFromFile (&movieB, resFref, &resID, (StringPtr)nil, 0,
    nil);
if (err) return;

CloseMovieFile (resFref);

```

Once the two movies have been loaded, the sample program retrieves the first track, which is the sprite track, from the original movie, and sets the selection to the start of the movie (Listing 3-14). The sample program iterates through all the tracks in the modifier movie, disposing of all non-video tracks.

Next, the sample program calls `AddMovieSelection` to add the modifier track to the original movie. Finally, the sample program calls `AddTrackReference` to associate the modifier track with the sprite track it will modify.

`AddTrackReference` returns an index of the added reference in the `referenceIndex` variable.

Listing 3-14 Adding the modifier track to the movie

```
Movie          m;
TimeValue      oldDuration;
Movie          movieB;
long           i, origTrackCount, referenceIndex;
Track          newTrack, spriteTrack;

// get the first track in original movie and position at the start
spriteTrack = GetMovieIndTrack (m, 1);
SetMovieSelection (m, 0 ,0);

// remove all tracks except video in modifier movie
for (i = 1; i <= GetMovieTrackCount (movieB); i++)
{
    Track t = GetMovieIndTrack (movieB, i);
    OSType aType;

    GetMediaHandlerDescription (GetTrackMedia(t), &aType, nil, nil);
    if (aType != VideoMediaType)
    {
        DisposeMovieTrack (t);
        i--;
    }
}

// add the modifier track to original movie
oldDuration = GetMovieDuration (m);
AddMovieSelection (m, movieB);
```

Authoring Wired Movies and Sprite Animations

```

DisposeMovie (movieB);

// truncate the movie to the length of the original track
DeleteMovieSegment (m, oldDuration,
    GetMovieDuration (m) - oldDuration);

// associate the modifier track with the original sprite track
newTrack = GetMovieIndTrack (m, origTrackCount + 1);
AddTrackReference (spriteTrack, newTrack, kTrackModifierReference,
    &referenceIndex);

```

Besides adding a reference to the modifier track, the sample program must update the sprite media's input map to describe how the modifier track should be interpreted by the sprite track. The sample program performs the following tasks (Listing 3-15):

- It retrieves the sprite track's media by calling `GetTrackMedia`.
- It calls `GetMediaInputMap` to retrieve the media's input map.
- It adds a parent atom to the input map of type `kTrackModifierInput`. The ID of the atom is the reference index retrieved by the `AddTrackReference` function.
- It adds two child atoms, one that specifies that the input type of the modifier track is of type `kTrackModifierTypeImage`, and one that specifies the index of the sprite image to override.
- It calls `SetMediaInputMap` to update the media's input map.

Listing 3-15 Updating the media's input map

```

#define kImageIndexToOverride 1

Movie          m, movieB;
long           referenceIndex, imageIndexToOverride;
Track          spriteTrack;
QTAtomContainer inputMap;
QTAtom         inputAtom;
OSType         inputType;
Media          spriteMedia;

```


Authoring Wired Movies and Sprite Animations

```
// get the sprite media's input map
spriteMedia = GetTrackMedia (spriteTrack);
GetMediaInputMap (spriteMedia, &inputMap);

// add an atom for a modifier track
QTInsertChild (inputMap, kParentAtomIsContainer,
               kTrackModifierInput, referenceIndex, 0, 0, nil, &inputAtom);

// add a child atom to specify the input type
inputType = kTrackModifierTypeImage;
QTInsertChild (inputMap, inputAtom, kTrackModifierType, 1, 0,
               sizeof(inputType), &inputType, nil);

// add a second child atom to specify index of image to override
imageIndexToOverride = kImageIndexToOverride;
QTInsertChild (inputMap, inputAtom, kSpritePropertyImageIndex, 1, 0,
               sizeof(imageIndexToOverride), &imageIndexToOverride, nil);

// update the sprite media's input map
SetMediaInputMap (spriteMedia, inputMap);
QTDisposeAtomContainer (inputMap);
```

Once the media's input map has been updated, the application can save the movie.

Authoring Wired Movies

In addition to providing functions that allow you to create and manipulate a sprite animation as a track in a QuickTime movie, the sprite media handler also provides functions that allow your application to create and manipulate a wired sprite movie, with various types of user interactivity.

The following sections are illustrated with code from the sample program `QTWiredSprites.c`, which shows how to create a sample wired sprite movie containing one sprite track. (A partial code listing is available in Appendix C. You can download the full sample code at QuickTime Web site at <http://www.apple.com/quicktime/developers/samplecode.html#sprites>.)

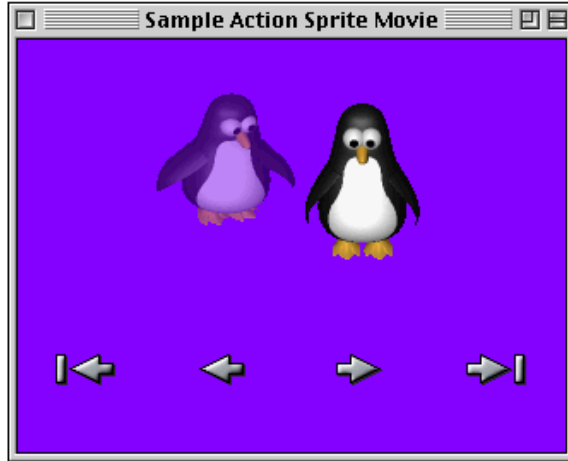
The sample code creates a 320 by 240 pixel wired movie with one sprite track that contains six sprites, two of which are penguins and four of which are buttons. Figure 3-1 shows two penguins at the outset of the movie, with the buttons invisible.

Figure 3-1 Two penguins from a sample program



Initially, the four buttons in the wired movie are invisible. When the mouse enters or “rolls over” a button, it appears, as shown in Figure 3-2.

Figure 3-2 Two penguins and four buttons, indicating various directions in the movie



When the mouse is clicked inside a button, its images change to its “pressed” image. When the mouse is released, its image is changed back to its “unpressed” image. If the mouse is released inside the button, it triggers an action. The buttons perform the following set of actions:

- go to beginning of movie
- step backwards
- step forwards
- go to end of movie

Actions of the First Penguin

The first penguin shows all of the buttons when the mouse enters it, and hides them when the mouse exits. The first penguin is the only sprite which has properties that are overridden by the override sprite samples. These samples override its matrix in order to move it, and its image index in order to make it waddle.

When you mouse-click on the second penguin, the penguin changes its image index to its “eyes closed” image. When the mouse is released, it changes back to

its normal image. This makes the penguin's eyes appear to blink when clicked on. When the mouse is released over the penguin, several other actions are triggered. Both penguins' graphics states are toggled between `copyMode` and `blendMode`, and the movie's rate is toggled between 0 and 1.

Actions of the Second Penguin

The second penguin moves once per second. This occurs whether the movie's rate is currently 0 or 1 because it is being triggered by a QuickTime idle event. When the penguin receives the idle event, it changes its matrix using an action which uses min, max, delta, and wraparound options.

The movie's looping mode is set to palindrome by a `kQTEventFrameLoaded` event.

Creating a Wired Sprite Movie

The following tasks are performed in order to create the wired sprite movie:

- You create a new movie file with a single sprite track, as explained in the section “Creating the Movie, Sprite Track, and Media” (page 50).
- You assign the “no controller” movie controller to the movie.
- You set the sprite track's background color, idle event frequency, and `hasActions` properties.
- You convert PICT resources to animation codec images with transparency.
- A key frame sample containing six sprites and all of their shared images is created. The sprites are assigned initial property values. A `frameLoaded` event is created for the key frame.
- You create some override samples which override the matrix and image index properties of the first penguin sprite.

Assigning the No Controller to the Movie

The following code fragment (in Listing 3-16) assigns the “no controller” movie controller to the movie. You make this assignment if you don't want the standard QuickTime movie controller to appear. In this code sample, you want to create a set of sprite buttons in order to control user interaction with the penguins in the movie.

There may also be other occasions when it is useful to make a “no controller” assignment. For example, if you are creating non-linear movies—such as Hypercard stacks where you access the cards in the stack by clicking on buttons by sprites—you may wish to create your own sprite buttons.

Listing 3-16 Assigning the no controller movie controller

```
// select the "no controller" movie controller
myType = EndianU32_NtoB(myType);
SetUserDataItem(GetMovieUserData(myMovie), &myType, sizeof(myType),
                kUserDataMovieControllerType, 1);
```

Setting Up the Sprite Track's Properties

Listing 3-17 shows a code fragment that sets the sprite track's background color, idle event frequency and its `hasActions` properties.

Listing 3-17 Setting the background color, idle event frequency and `hasActions` properties of the sprite track

```
// set the sprite track properties
{
    QTAtomContainer    myTrackProperties;
    RGBColor           myBackgroundColor;

    // add a background color to the sprite track
    myBackgroundColor.red = EndianU16_NtoB(0x8000);
    myBackgroundColor.green = EndianU16_NtoB(0);
    myBackgroundColor.blue = EndianU16_NtoB(0xffff);

    QTNewAtomContainer(&myTrackProperties);
    QTInsertChild(myTrackProperties, 0,
                 kSpriteTrackPropertyBackgroundColor, 1, 1,
                 sizeof(RGBColor), &myBackgroundColor, NULL);

    // tell the movie controller that this sprite track has actions
    hasActions = true;
    QTInsertChild(myTrackProperties, 0,
                 kSpriteTrackPropertyHasActions, 1, 1,
```

```

        sizeof(hasActions), &hasActions, NULL);

    // tell the sprite track to generate QTIdleEvents
    myFrequency = EndianU32_NtoB(60);
    QTInsertChild(myTrackProperties, 0,
                  kSpriteTrackPropertyQTIdleEventsFrequency, 1, 1,
                  sizeof(myFrequency), &myFrequency, NULL);
    myErr = SetMediaPropertyAtom(myMedia, myTrackProperties);
    if (myErr != noErr)
        goto bail;

    QTDisposeAtomContainer(myTrackProperties);
}

```

Adding an Event Handler to the Penguin

The `AddPenguinTwoConditionalActions` routine adds logic to our penguin. Using this routine, you can transform the penguin into a two-state button that plays/pauses the movie.

We are relying on the fact that a `GetVariable` for a `variableID` which has never been set will return 0. If we need another default value, we could initialize it using the `frameLoaded` event.

A higher level description of the logic is:

```

On MouseUpInside
    If (GetVariable(DefaultTrack, 1) = 0)
        SetMovieRate(1)
        SetSpriteGraphicsMode(DefaultSprite, { blend, grey })
        SetSpriteGraphicsMode(GetSpriteByID(DefaultTrack, 5), {
                                ditherCopy, white })
        SetVariable(DefaultTrack, 1, 1)
    ElseIf (GetVariable(DefaultTrack, 1) = 1)
        SetMovieRate(0)
        SetSpriteGraphicsMode(DefaultSprite, { ditherCopy, white })
        SetSpriteGraphicsMode(GetSpriteByID(DefaultTrack, 5), { blend,
                                                                grey })
        SetVariable(DefaultTrack, 1, 0)
    Endif
End

```

Adding a Series of Actions to the Penguins

The following code fragment in Listing 3-18 shows how you can add a key frame with four buttons, enabling our penguins to move through a series of actions.

Listing 3-18 Adding a key frame with four buttons, enabling a series of actions for our two penguins

```
// add actions to the six sprites
// add go to beginning button
myErr = QTCopyAtom(myBeginButton, kParentAtomIsContainer,
                  &myBeginActionButton);
if (myErr != noErr)
    goto bail;

AddSpriteSetImageIndexAction(myBeginActionButton,
                             kParentAtomIsContainer, kQTEventMouseClicked, 0, NULL,
                             0, 0, NULL, kGoToBeginningButtonDownIndex, NULL);
AddSpriteSetImageIndexAction(myBeginActionButton,
                             kParentAtomIsContainer, kQTEventMouseClickedEnd, 0,
                             NULL, 0, 0, NULL, kGoToBeginningButtonUpIndex, NULL);
AddSpriteToSample(mySample, myPrevActionButton, kGoToPrevSpriteID);
NULL);
AddSpriteToSample(mySample, myNextActionButton, kGoToNextSpriteID);
QTDisposeAtomContainer(myNextActionButton);
. . .
// add go to end button
myErr = QTCopyAtom(myEndButton, kParentAtomIsContainer,
                  &myEndActionButton);
if (myErr != noErr)
    goto bail;
    kParentAtomIsContainer, kQTEventMouseExit, 0, NULL, 0, 0, NULL,
    false,
. . .

// add penguin one
myErr = QTCopyAtom(myPenguinOne, kParentAtomIsContainer,
                  &myPenguinOneAction);
if (myErr != noErr)
```

Authoring Wired Movies and Sprite Animations

```

AddSpriteSetVisibleAction(myBeginActionButton, goto bail;

// show the buttons on mouse enter and hide them on mouse exit
AddSpriteSetVisibleAction(myPenguinOneAction, kParentAtomIsContainer,
                           kQTEventMouseEnter, 0, NULL, 0,
                           kTargetSpriteID, (void
                           *)kGoToBeginningSpriteID, true, NULL);
AddSpriteSetVisibleAction(myPenguinOneAction, kParentAtomIsContainer,
                           kQTEventMouseExit, 0, NULL, 0,
                           kTargetSpriteID, (void
                           *)kGoToBeginningSpriteID, false, NULL);
. . .
// add penguin two
myErr = QTCopyAtom(myPenguinTwo, kParentAtomIsContainer,
                  &myPenguinTwoAction);

if (myErr != noErr)
    goto bail;

// blink when clicked on
AddSpriteSetImageIndexAction(myPenguinTwoAction,
                             kParentAtomIsContainer,
                             kQTEventMouseClicked, 0, NULL, 0, 0, NULL,
                             kPenguinClosedIndex, NULL);
. . .
// add go to next button
myErr = QTCopyAtom(myNextButton, kParentAtomIsContainer,
                  &myNextActionButton);

if (myErr != noErr)
    goto bail;

```

Important Things to Note in the Sample Code

You should note the following in `MakeActionSpriteMovie.c` sample code:

- There are event types other than mouse-related events (for example, `Idle` and `frameLoaded`).
- Idle events are independent of the movie's rate, and can be gated, so they are sent at most every `n` ticks. (The second penguin moves when the movie's rate is 0, and moves only once per second because of the value of the sprite tracks `idleEventFrequency` property.)

- Multiple actions may be executed in response to a single event (for example, rolling over the first penguin shows and hides four different buttons).
- Actions may target any sprite or track in the movie (for example, clicking on one penguin changes the graphics mode of the other).
- Conditional and looping control structures are supported. (The second penguin uses the “case statement” action.)
- Sprite track variables that have not been set have a default value of 0. (The second penguin’s conditional code relies on this.)

New Authoring Features in QuickTime 4

QuickTime 4 enables you to author movies with external movie targets. To accomplish this, two new target atoms have been introduced, as explained in the following section.

Specifying an External Movie Target for an Action

In order to specify that an action is to target an element of an external movie, the external movie must be identified by either its name or its ID. To do this, two new target atom types have been introduced; these atoms are used in addition to the existing target atoms, which may specify that the element is a particular Track or object within a Track such as a Sprite.

Note

A movie ID may be specified by an expression. ♦

These are the additional target atoms provided in QuickTime 4:

```
[(ActionTargetAtoms)] =
    <kActionTarget>

        <kTargetMovieName>
            [Pstring MovieName]
        OR
        <kTargetMovieID>
```

```
[long MovieID]
OR
[(kExpressionAtoms)]
```

Tagging a Movie with a Name or ID

To tag a movie with a Name or ID, you add a user data item of type ‘plug’ to the movie’s user data. The index of the user data does not matter. The data specifies the Name or ID.

Naming a Movie

You add a user data item of type ‘plug’ to the Movie’s user data with its data set to

```
"Movieid=MovieName"
```

where `MovieName` is the name of the Movie.

Setting a Movie’s ID

You add a user data item of type ‘plug’ to the Movie’s user data with its data set to

```
"Movieid=MovieID"
```

where the ID is a signed long integer.

The QuickTime plug-in additionally supports EMBED tag parameters which allow you to override a movie’s Name or ID within an HTML page.

Supporting External Movie Targets in Your Application

If you want your application to support external movie targets, you need to resolve the movie names or IDs to actual movie references. This is accomplished by installing a Movie Controller filter proc for each Movie Controller that filters for the `mcActionGetExternalMovie` action.

The parameter to the `mcActionGetExternalMovie` Movie Controller action is a pointer to a `QTGetExternalMovieRecord`.

First, you look at the `targetType` field. If it is set to `kTargetMovieName`, then return the movie and Movie Controller for the movie named by the `MovieName` field. If it is set to `kTargetMovieID`, then return the movie and Movie Controller for the movie with ID equal to the `MovieID` field.

If you cannot find a matching movie, then set `theMovie` and `theController` to `nil`.

New Grammar for Specifying String Parameters to Actions and Operands

QuickTime 3 supported only leaf data constants for string parameters. In QuickTime 4, you can specify string parameters using leaf data constants, a numeric expression, or a sprite track variable.

Expressions are converted to strings, as are sprite track variables if they contain a floating-point number instead of a string. Note that although string variables are stored as C strings, they will be automatically converted to Pascal strings as needed.

This applies to the following action and operand parameters:

C String Parameters

```
kActionGoToURL, 1
kActionStatusString, 1
kActionAddChannelSubscription, 2
kActionAddChannelSubscription, 3
kActionRemoveChannelSubscription, 1
kOperandSubscribedToChannel, 1
```

Pascal String Parameters

```
kActionMovieGoToTimeByName, 1
kActionMovieSetSelectionByName, 1
kActionMovieSetSelectionByName, 2
kActionPushCurrentTimeWithLabel, 1
kActionPopAndGotoLabeledTime, 1
kActionDebugStr, 1
kActionAddChannelSubscription, 1
```

Extended Grammar:

```
[(C String Parameter)] =
```

```

        [CString]
    OR
        [(ExpressionAtoms)]
    OR
        [(SpriteTrackVariableOperandAtoms)]

[(Pascal String Parameter)] =
    [Pascal String]
    OR
        [(ExpressionAtoms)]
    OR
        [(SpriteTrackVariableOperandAtoms)]

[(SpriteTrackVariableOperandAtoms)] =
    <kOperandSpriteTrackVariable>, 1, 1
        [(ActionTargetAtoms)]
        kActionParameter, 1, 1
            [(spriteVariableID)]
            OR
            [(ExpressionAtoms)]

```

Wiring a Movie by Adding an HREF Track

QuickTime enables you to wire a movie by adding an **HREFTrack**, which causes the QuickTime plug-in to load URLs as a movie plays or in response to user actions.

About HREFTracks

An HREFTrack is a specially named text track that contains hypertext references (HREFs). These references are in the form of URLs and can point to any kind of data that a URL can specify, such as a Web page, a QuickTime movie, or a JavaScript.

A URL can be invoked automatically when the movie plays, or it can be invoked interactively when the user clicks the mouse inside a movie. The URL

can optionally be targeted to a named frame or browser window, or to the QuickTime movie itself.

HREFTracks are only active when the movie is played by the QuickTime plug-in. The standard movie controller and the QuickTime Player application treat HREFTracks as ordinary text tracks.

Because an HREFTrack is a text track, it will display its samples as on-screen text while the movie plays. This is normally desirable during debugging. To turn the text off, disable the track by calling the `SetTrackEnabled` function.

When flattening a movie that contains an HREFTrack, be careful not to accidentally delete the disabled text track by calling `FlattenMovie` with the `flattenActiveTracksOnly` flag.

You can only have one HREFTrack per movie. If you add more than one HREFTrack to a movie, the QuickTime Plugin will use the first one it finds and ignore any others.

HREFTrack Syntax

The syntax for an HREFTrack sample is:

```
[nn:nn:nn.n]  
A<URL> T<frame>
```

`[nn:nn:nn.n]` is a timestamp in hours:minutes:seconds.timeunits, where timeunits are in the time scale of the track. The URL becomes active at the time specified and remains active until the next timestamp. The URL is active even if the movie is not playing.

The last specified URL remains active even after the movie has finished.

`A<URL>` specifies the URL. If the “A” is omitted, the user must click inside the movie during the period when the URL is active to cause the URL to load. If the “A” is present, the URL will load automatically when the movie is played.

The URL can be absolute or relative to the movie.

IMPORTANT

A relative URL is relative to the movie, not the Web page that contains the movie. If the movie and its parent Web page are in the same directory, this is the same thing, but if the movie and the Web page are not in the same directory, the difference is crucial. ▲

The URL can include an internal anchor name as well as a path and file name (for example, “../HTML/Page1.htm#StepOne”).

The URL can be a blank sample (CRLF). Clicking inside the movie when a blank sample is active has no effect. Use a blank sample to deactivate the previous URL without activating a new one. You might commonly want to do this at the end of a movie.

`T<frame>` is an optional parameter. It specifies a target frame or target window. If a target is specified, the URL will be loaded in the specified frame or window. If there is no frame or window with that name, a new browser window of that name will be created.

You can combine the `T<frame>` parameter with the “A” in the URL to cause a movie playing in one frame to automatically load a series of URLs in another frame at specified points in the movie.

If no target is specified, the URL will load in the default browser window or browser frame. In this case, the Web page containing the movie will be replaced by the specified URL.

If the target “myself” is specified, the URL will be opened directly by the QuickTime plug-in. The specified URL will replace the currently-playing movie, rather than its parent Web page. This will only work properly if the URL points to a file that QuickTime can handle directly, such as a QuickTime movie or an AIFF audio file.

You can use the `T<myself>` parameter to link a series of QuickTime movies into a single virtual movie over the Web.

Creating an HREFTrack

Like any text track, an HREFTrack can be created using a text editor and then imported using a movie import component, or it can be created programmatically.

You create an HREFTrack programmatically by creating a text track, adding the URLs as text samples, then setting the name of the track to “HREFTrack”. The track name is stored in the user data atom.

The following code snippet shows how to change the name of a text track to “HREFTrack”.

```
// create a Pascal string 'HREFTrack'
Str255 trackNameStr = " HREFTrack" // leave a space for length byte
trackNameStr[0] = strlen(trackNameStr) - 1; // add length byte
// store it as the track name
UserDat ud = GetTrackUserData(theTrack);
OSType dataType = FOUR_CHAR_CODE('name');

RemoveUserData(ud, dataType, 1); // remove existing name, if any
SetUserDataItem(ud, &trackNameStr[1], trackNameStr[0], dataType, 0);
```

CHAPTER 3

Authoring Wired Movies and Sprite Animations

Using the Sprite Toolbox to Create Sprite Animations

This chapter discusses the **sprite toolbox** which is a set of data types and functions you can use to add sprite-based animation to an application. The sprite toolbox handles the following:

- invalidating appropriate areas as sprite properties change
- the composition of sprites and their background on an offscreen buffer
- the transfer of the result to the screen or to an alternate destination.

If you're authoring an animation *outside* of a movie, you use the sprite toolbox to create sprite worlds and sprite animations. For more information about the constants and data types available to your application in the sprite toolbox, refer to Chapter 7, "Sprite Toolbox API Reference."

To create a sprite track *in* a QuickTime movie, you create media samples used by the sprite media handler, which, in turn, makes use of the sprite toolbox. For information on how to use the sprite media handler, see Chapter 2, "The Sprite Media Handler."

This chapter is aimed at developers who are using the lower-level sprite toolbox APIs to create sprite animations in their applications, not in a QuickTime movie.

How To Add Sprite-Based Animations to an Application

The following section discusses how you can use the sprite toolbox to create sprite worlds and sprite animations. It is divided into these topics:

- "Creating and Initializing a Sprite World" (page 82)
- "Creating and Initializing Sprites" (page 84)
- "Animating Sprites" (page 87)

- “Disposing of a Sprite Animation” (page 89)
- “Sprite Hit Testing” (page 90)
- “Enhancing Sprite Animation Performance” (page 91)

Creating and Initializing a Sprite World

To create a sprite animation in an application, you first create a sprite world to contain your sprites. To do this, you perform the following steps:

- Allocate a sprite layer graphics world that corresponds to the size and bit depth of your destination graphics world.
- If you plan to have a background image behind your sprites that is static or that changes infrequently, create a background graphics world that is the same size and depth as the sprite layer graphics world. You do not need to do this if you plan to have a solid background color behind your sprites. Animations that use a solid background color require less memory and perform slightly better than animations that use a background image.
- Call `LockPixels` on the pixel maps of the sprite layer and background graphics worlds. These graphics worlds must remain valid for the lifetime of the sprite world.
- Call the `NewSpriteWorld` function to create the new sprite world.

The sample code function `CreateSpriteStuff`, shown in Listing 4-1, calculates the bounds of the destination window and calls `NewGWorld` to create a new sprite layer graphics world. It then calls `LockPixels` to lock the pixel map of the sprite layer graphics world.

Next, `CreateSpriteStuff` calls `NewSpriteWorld` to create a new sprite world, passing the destination graphics world (`WindowPtr`) and the sprite layer graphics world. `CreateSpriteStuff` passes a background color to `NewSpriteWorld` instead of specifying a background graphics world. The newly created sprite world is returned in the global variable `gSpriteWorld`.

Finally, `CreateSpriteStuff` calls the sample code function `CreateSprites` to populate the sprite world with sprites.

Using the Sprite Toolbox to Create Sprite Animations

Listing 4-1 Creating a sprite world

```

// global variables
GWorldPtr spritePlane = nil;
SpriteWorld gSpriteWorld = nil;
Rect gBounceBox;
RGBColor gBackgroundColor;

void CreateSpriteStuff (Rect *windowBounds, CGrafPtr windowPtr)
{
    OSErr err;
    Rect bounds;

    // calculate the size of the destination
    bounds = *windowBounds;
    OffsetRect (&bounds, -bounds.left, -bounds.top);
    gBounceBox = bounds;
    InsetRect (&gBounceBox, 16, 16);

    // create a sprite layer graphics world with a bit depth of 16
    NewGWorld (&spritePlane, 16, &bounds, nil, nil, useTempMem);
    if (spritePlane == nil)
        NewGWorld (&spritePlane, 16, &bounds, nil, nil, 0);

    if (spritePlane)
    {
        LockPixels (spritePlane->portPixMap);
        gBackgroundColor.red = gBackgroundColor.green =
            gBackgroundColor.blue = 0;

        // create a sprite world
        err = NewSpriteWorld (&gSpriteWorld, (CGrafPtr>windowPtr,
            spritePlane, &gBackgroundColor, nil);

        // create sprites
        CreateSprites ();
    }
}

```

Creating and Initializing Sprites

Once you have created a sprite world, you can create sprites within it. To do this, you must first obtain image descriptions and image data for your sprites. This image data may be any image data that has been compressed using QuickTime's Image Compression Manager.

You create sprites and add them to your sprite world using the `NewSprite` function. If you want to create a sprite that is drawn to the background graphics world, you should specify the constant `kBackgroundSpriteLayerNum` for the `layer` parameter.

Note

The compressed image data must remain locked as long as it is set to be the sprite's image data. ♦

Creating Sprites for a Sample Application

The sample code function `CreateSprites`, shown in Listing 4-2, creates the sprites for the sample application shown in Listing 4-1.

First, the function initializes some global arrays with position and image information for the sprites. Next, `CreateSprites` iterates through all the sprite images, preparing each image for display. For each image, `CreateSprites` calls the sample code function `MakePictTransparent` function, which strips any surrounding background color from the image. `MakePictTransparent` does this by using the animation compressor to recompress the PICT images using a key color. Then, `CreateSprites` calls `ExtractCompressData`, which extracts the compressed data from the PICT image. This is one technique for creating compressed images; there are other, more optimized ways to store and retrieve sprite images.

Once the images have been prepared, `CreateSprites` calls `NewSprite` to create each sprite in the sprite world. `CreateSprites` creates each sprite in a different layer.

Listing 4-2 Creating sprites

```
// constants
#define kNumSprites 4
#define kNumSpaceShipImages 24
```

Using the Sprite Toolbox to Create Sprite Animations

```

#define kBackgroundPictID 158
#define kFirstSpaceShipPictID (kBackgroundPictID + 1)
#define kSpaceShipWidth 106
#define kSpaceShipHeight 80

// global variables
SpriteWorld gSpriteWorld = nil;
Sprite gSprites[kNumSprites];
Rect gDestRects[kNumSprites];
Point gDeltas[kNumSprites];
short gCurrentImages[kNumSprites];
Handle gCompressedPictures[kNumSpaceShipImages];
ImageDescriptionHandle gImageDescriptions[kNumSpaceShipImages];

void CreateSprites (void)
{
    long i;
    Handle compressedData = nil;
    PicHandle picture;
    CGrafPtr savePort;
    GDHandle saveGD;
    OSErr err;
    RGBColor keyColor;

    SetRect (&gDestRects[0], 132, 132, 132 + kSpaceShipWidth,
            132 + kSpaceShipHeight);
    SetRect (&gDestRects[1], 50, 50, 50 + kSpaceShipWidth,
            50 + kSpaceShipHeight);
    SetRect (&gDestRects[2], 100, 100, 100 + kSpaceShipWidth,
            100 + kSpaceShipHeight);
    SetRect (&gDestRects[3], 130, 130, 130 + kSpaceShipWidth,
            130 + kSpaceShipHeight);

    gDeltas[0].h = -3;
    gDeltas[0].v = 0;
    gDeltas[1].h = -5;
    gDeltas[1].v = 3;
    gDeltas[2].h = 4;
    gDeltas[2].v = -6;
    gDeltas[3].h = 6;
    gDeltas[3].v = 4;

```

Using the Sprite Toolbox to Create Sprite Animations

```

gCurrentImages[0] = 0;
gCurrentImages[1] = kNumSpaceShipImages / 4;
gCurrentImages[2] = kNumSpaceShipImages / 2;
gCurrentImages[3] = kNumSpaceShipImages * 4 / 3;

keyColor.red = keyColor.green = keyColor.blue = 0xFFFF;

// recompress PICT images to make them transparent
for (i = 0; i < kNumSpaceShipImages; i++)
{
    picture = (PicHandle) GetPicture (i + kFirstSpaceShipPictID);
    DetachResource ((Handle)picture);

    MakePictTransparent (picture, &keyColor);
    ExtractCompressData (picture, &gCompressedPictures[i],
        &gImageDescriptions[i]);
    HLock (gCompressedPictures[i]);

    KillPicture (picture);
}

// create the sprites for the sprite world
for (i = 0; i < kNumSprites; i++)
{
    MatrixRecord matrix;

    SetIdentityMatrix (&matrix);

    matrix.matrix[2][0] = ((long)gDestRects[i].left << 16);
    matrix.matrix[2][1] = ((long)gDestRects[i].top << 16);

    err = NewSprite (&(gSprites[i]), gSpriteWorld,
        gImageDescriptions[i],* gCompressedPictures[i],
        &matrix, true, i);
}
}

```

Animating Sprites

To animate a sprite, you use the `SetSpriteProperty` function to change one or more of the sprite's properties, such as its matrix, layer, or image data. In addition to modifying a property, `SetSpriteProperty` invalidates the appropriate areas of the sprite's sprite world.

The `SpriteWorldIdle` function is responsible for redrawing a sprite world's invalid regions. Your application should call this function after modifying sprite properties to give the sprite world the opportunity to redraw.

Listing 4-3 shows the sample application's `main` function. It performs all of the application's initialization tasks, including initializing the sprite world and its sprites. It displays the window and loops until the user clicks the button in the window. To perform the animation, `main` calls the sample code function `MoveSprites` each time through the loop, to modify the properties of the sprites, and then calls `SpriteWorldIdle` to give the sprite world the opportunity to redraw its invalid areas.

Listing 4-3 The `main` function

```
// global variables
SpriteWorld gSpriteWorld = nil;

void main (void)
{
    // ...
    // initialize everything and create a window
    // create a sprite world and the sprites in it
    // show the window
    // ...
    CreateSpriteStuff(...);
    while (!Button())
    {
        // animate the sprites
        MoveSprites ();
        SpriteWorldIdle (gSpriteWorld, 0, 0);
    }

    // ...
}
```

Using the Sprite Toolbox to Create Sprite Animations

```

        // dispose of the sprite world and its sprites
        // shut down everything else
        // ...
        DisposeEverything();
    }

```

The `MoveSprites` function, shown in Listing 4-4, is responsible for modifying the properties of the sprites. For each sprite, the function calls `SetSpriteProperty` twice, once to change the sprite's matrix and once to change the sprite's image data pointer.

Listing 4-4 Animating sprites

```

// constants
#define kNumSprites 4
#define kNumSpaceShipImages 24

// global variables
Rect gBounceBox;
Sprite gSprites[kNumSprites];
Rect gDestRects[kNumSprites];
Point gDeltas[kNumSprites];
short gCurrentImages[kNumSprites];
Handle gCompressedPictures[kNumSpaceShipImages];

void MoveSprites (void)
{
    short i;
    MatrixRecord matrix;

    SetIdentityMatrix (&matrix);

    // for each sprite
    for (i = 0; i < kNumSprites; i++)
    {
        // modify the sprite's matrix
        OffsetRect (&gDestRects[i], gDeltas[i].h, gDeltas[i].v);

        if ( (gDestRects[i].right >= gBounceBox.right) ||
            (gDestRects[i].left <= gBounceBox.left) )

```


Using the Sprite Toolbox to Create Sprite Animations

```

        gDeltas[i].h = -gDeltas[i].h;

        if ( (gDestRects[i].bottom >= gBounceBox.bottom) ||
            (gDestRects[i].top <= gBounceBox.top) )
            gDeltas[i].v = -gDeltas[i].v;

        matrix.matrix[2][0] = ((long)gDestRects[i].left << 16);
        matrix.matrix[2][1] = ((long)gDestRects[i].top << 16);

        SetSpriteProperty (gSprites[i], kSpritePropertyMatrix, &matrix);

        // change the sprite's image
        gCurrentImages[i]++;
        if (gCurrentImages[i] >= (kNumSpaceShipImages * (i+1)))
            gCurrentImages[i] = 0;
        SetSpriteProperty (gSprites[i], kSpritePropertyImageDataPtr,
            *gCompressedPictures[gCurrentImages[i] / (i+1)] );
    }
}

```

Disposing of a Sprite Animation

When your application has finished displaying a sprite animation, you should do the following things in the order shown:

1. Dispose of the sprite world associated with the animation. (You need to do this first.) Disposing of a sprite world automatically destroys the sprites in the sprite world.
2. Dispose of the sprite image data.
3. Dispose of graphics worlds associated with the sprite animation.

In the sample application, `main` calls the sample code function

`DisposeEverything` to dispose of sprite-related structures. This function, shown in Listing 4-5, iterates through the sprites, disposing of each sprite's image data. Then, `DisposeEverything` calls `DisposeSpriteWorld` to dispose of the sprite world and all of the sprites in it. Finally, the function calls `DisposeGWorld` to dispose of the graphics world associated with the sprite world.

Listing 4-5 Disposing of sprites and the sprite world

```

// constants
#define kNumSprites 4
#define kNumSpaceShipImages 24

// global variables
SpriteWorld gSpriteWorld = nil;
Sprite gSprites[kNumSprites];
Handle gCompressedPictures[kNumSpaceShipImages];
ImageDescriptionHandle gImageDescriptions[kNumSpaceShipImages];

void DisposeEverything (void)
{
    short i;
    // dispose of the sprite world and associated graphics world
    if (gSpriteWorld)
        DisposeSpriteWorld (gSpriteWorld);

    // dispose of each sprite's image data
    for (i = 0; i < kNumSprites; i++)
    {
        if (gCompressedPictures[i])
            DisposeHandle (gCompressedPictures[i]);
        if (gImageDescriptions[i])
            DisposeHandle ((Handle)gImageDescriptions[i]);
    }
    DisposeGWorld (spritePlane);
}

```

Sprite Hit Testing

The sprite toolbox provides two functions for performing hit testing operations with sprites, `SpriteWorldHitTest` and `SpriteHitTest`.

The `SpriteWorldHitTest` function determines whether any sprites exist at a specified location in a sprite world's display coordinate system. This function retrieves the frontmost sprite at the specified location.

The `SpriteHitTest` function determines whether a particular sprite exists at a specified location in the sprite's display coordinate system. This function is

useful for hit testing a subset of the sprites in a sprite world and for detecting multiple sprites at a single location.

For either hit test function, there are two flags, `spriteHitTestBounds` and `spriteHitTestImage`, that control the hit test operation. For example, you set the `spriteHitTestBounds` flag to check if there has been a hit anywhere within the sprite's bounding box, and you set the `spriteHitTestImage` flag to check if there has been a hit anywhere within the sprite image.

These flags are described in “Sprite Hit Testing” (page 90).

Hit Testing Flags

The following hit testing flags are included in QuickTime 3. These flags are used with both the sprite toolbox and the movie sprite track hit testing routines:

- `spriteHitTestInvisibleSprites`, which you set if you want invisible sprites to be hit tested along with visible ones.
- `spriteHitTestLocInDisplayCoordinates`, which you set if the hit testing point is in display coordinates instead of local sprite track coordinates.
- `spriteHitTestIsClick`, which you set if you want the hit testing operation to pass a click on to the codec currently rendering the sprites image. For example, this can be used to make the Ripple Codec ripple.

Enhancing Sprite Animation Performance

To achieve the best performance for your sprite animation, you should observe the following guidelines when creating a sprite world.

- When you create a graphics world to be used for your sprite world, you achieve the best performance if the graphics world's dimensions are a multiple of 16 pixels.
- Your sprite layer graphics world and background graphics world should both be the same size and depth as the destination of your sprite animation.
- Use translation-only matrices for creating sprite worlds and sprites.
- Do not set a clipping region for your sprite world.
- Call the `SpriteWorldIdle` function frequently.
- Avoid clipping sprites with the sprite world boundary.

CHAPTER 4

Using the Sprite Toolbox to Create Sprite Animations

- Use the Animation compressor to create sprites with transparent areas.

Flash Media Handler

This chapter describes the **Flash media handler**, which is new in QuickTime 4 and allows a Macromedia Flash SWF 3.0 file to be treated as a track within a QuickTime movie. QuickTime 4 extends the SWF file format to allow the execution of any of its wired actions.

The chapter also discusses how you can add wired actions to a Flash track.

Support For Macromedia's Flash in QuickTime 4

Introduced in 1996 by Macromedia, Flash is a vector-based graphics and animation technology designed for the Internet. As an authoring tool, Flash lets content authors and developers create a wide range of interactive vector animations. The files exported by this tool are called SWF (pronounced 'swiff') files.

SWF files are commonly played back using Macromedia's ShockWave Plugin. In an effort to establish Flash as an industry-wide standard, Macromedia has published the SWF File Format and made the specification publicly available on its Web site at <http://www.macromedia.com/software/flash/open/spec/>.

The Flash Media Handler

QuickTime 4 has added support for the interactive playback of SWF 3.0 files by introducing a new Flash media handler. (Note that future versions of QuickTime will support later versions of SWF files.) This media handler allows a SWF file to be treated as a track within a QuickTime movie. Because a QuickTime movie may contain any number of tracks, multiple SWF tracks may be added to the same movie. The Flash Media Handler also provides support

for an optimized case using the alpha channel graphics mode, which allows a Flash track to be composited cleanly over other tracks.

QuickTime supports all Flash actions except for the Flash load movie action. For example, when a Flash track in a QuickTime movie contains an action that goes to a particular Flash frame, QuickTime converts this to a wired action that goes to the QuickTime movie time in the corresponding Flash frame.

Note

As a time-based media playback format, QuickTime may drop frames when necessary to maintain its schedule. As a consequence, frames of a SWF file may be dropped during playback. If this is not satisfactory for your application, you may set the playback mode of the movie to Play All Frames, which will emulate the playback mode of ShockWave. QuickTime's SWF file importer sets the Play All Frames mode automatically when adding a SWF file to an empty movie. ♦

How To Add Wired Actions To a Flash Track

The sample code on the QuickTime SDK, `AddFlashActions`, provides the code you need to add wired actions to a Flash track. It may also be useful to download the Macromedia SWF File Format Specification at <http://www.macromedia.com/software/flash/open/spec/>, as well as the SWF File Parser code at the Macromedia Web site.

This section explains the steps you need to follow in order to add wired actions to a SWF 3.0 file.

Extending the SWF Format

QuickTime 4 extends the SWF file format to allow the execution of any of its wired actions, in addition to the much smaller set of Flash actions. For example, you may use a SWF file as a user interface element in a QuickTime movie, controlling properties of the movie and other tracks. QuickTime also allows SWF files to be compressed using the zlib data compressor. This can significantly lower the bandwidth required when downloading a SWF file when it is in a QuickTime movie.

By using wired actions within a Flash track, compressing your Flash tracks, and combining Flash tracks with other types of QuickTime media, you can create compact and sophisticated multimedia content.

The SWF File Format Specification consists of a header followed by a series of tagged data blocks. The types of tagged data blocks you need to use are the `DefineButton2` and `DoAction`. The `DefineButton2` block allows Flash actions to be associated with a mouse state transition. The `DoAction` allows actions to be executed when the tag is encountered. These are analogous to mouse-related QT Event handlers and the Frame Loaded event in wired movies.

Flash actions are stored in an `ActionRecord`. Each Flash action has its own tag, such as `ActionPlay` and `ActionNextFrame`. QuickTime defines one new tag: `QuickTimeActions`, which is `0xAA`. The data for the `QuickTimeActions` tag is simply a `QTAtomContainer` with the QuickTime wired actions to execute in it.

There are also fields you need to change in order to add wired actions to a SWF file. Additionally, there is one tag missing from the SWF file format that is described below.

What You Need to Modify

DefineButton2

For `defineButton2`, you need to modify or add the following fields:

File Length, `ActionRecordsOffset` (which is currently undocumented), the `ActionOffset`, the `Condition`, the `RecordHeader` size portion, and add `ActionRecord`.

File Length

A 32 bit field in the SWF file header.

`RecordHeader` for the `defineButton2`

The `RecordHeader` contains the `tagID` and length. You need to update the length. Note that there are short and long formats for record headers, depending on the size of the record. The `tagID` for `defineButton2` is 34.

ActionRecordsOffset

This 16 bit field is missing from the SWF File Format Specification. It occurs in between the `Flags` and `Buttons` fields. It is initially set to zero if there are no actions for the button. If there are actions for the button, then it must contain the offset from the point in the SWF file following this 16-bit value to the beginning of the `ActionOffset` field.

```
DefineButton2 =

    Header
    ButtonID
    Flags

    ActionRecordsOffset      (this is missing from the spec)

    Buttons
    ButtonEndFlag
    Button2ActionCode
    ActionOffset
    Condition
    Action                    [ActionRecords]
    ActionEndFlag
```

ActionOffset

There is one `ActionOffset` per condition (mouse `overDownToIdle`). This is the offset used to skip over the `Condition` and the following actions (the `ActionRecord`) for the condition. You need to update this value when adding actions.

Condition

The condition field is roughly equivalent to a wired movie event. The actions associated with button state transition condition are triggered when the transition occurs. You need to add or edit this field.

Actions

Flash actions each have their own action tag code. QuickTime actions use a single `QuickTimeActions` code: 'AA'. You may add a list of actions to a single `QuickTimeActions` tag.

The format of the QuickTimeActions tag is as follows:

1 byte:	Tag = 'AA'
2 bytes:	data length (size of the QTAtomContainer)
n bytes	the data which is the QTAtomContainer holding the wired actions

DoAction

For `DoAction`, you need to modify a subset of the `defineButton2` fields in the same manner as described above. These fields are:

File Length, the RecordHeader size portion, and the ActionRecord.

Endian Issues

You need to write the length fields in little endian format.

CHAPTER 5

Flash Media Handler

Wired Movies API Reference

This chapter provides an extensive API reference of constants and data types available to your application for wired movies support.

This chapter also describes the new movie actions and operands (page 107) available to your application in QuickTime 4.

For more conceptual information about wired movies, refer to Chapter 1, “Introduction to Wired Movies, Sprites, and the Sprite Toolbox.” For information about how to work with and author wired movies and sprite animations, refer to Chapter 3, “Authoring Wired Movies and Sprite Animations.”

Constants

Event Constants

Sprite samples are QTAtomContainers with `kSpriteAtomType` child atoms at the root level. The new `kQTEventType` atom is inserted as a child of a `kSpriteAtomType` atom in order to create an event handler for the given sprite. All other atom types are inserted into a `kQTEventType` atom or one of its descendents. There is one exception: the `kQTEventFrameLoaded` atom is inserted as a sibling of the `kSpriteAtomType` atoms, since the frame loaded event is associated with the whole sprite sample, not with any particular sprite.

Constant descriptions

<code>kQTEventType</code>	This atom is a container for a single type of QuickTime event handler. For a given sprite, you add one atom of this type for each type of QuickTime event that you want it to handle. The ID of the <code>kQTEventType</code> atom specifies the type
---------------------------	---

of `QuickTime` event. This atom's parent type is `kSpriteAtomType`.

All events are added to the sample using the `kQTEventType` atom, except for the frame loaded event.

`kQTEventMouseClicked`

Event sent to a sprite when the mouse is pressed down over it. This sprite becomes the one that receives the `kQTEventMouseClicked` event when the mouse button is released. If the mouse is released over the sprite, it will also receive the `kQTEventMouseClickedTriggerButton` event. You set the ID of a `kQTEventType` atom to `kQTEventMouseClicked` to create a handler for it.

`kQTEventMouseClickedEnd`

Event sent to the sprite that received the last `kQTEventMouseClicked` event when the mouse button is released. This event is sent regardless of where the current mouse location is. You set the ID of a `kQTEventType` atom to `kQTEventMouseClickedEnd` to create a handler for it.

`kQTEventMouseClickedEndTriggerButton`

Event sent to the sprite that received the last `kQTEventMouseClicked` event when the mouse button is released over the sprite. This event is sent in addition to the `kQTEventMouseClickedEnd` event, not instead of it. You set the ID of a `kQTEventType` atom to `kQTEventMouseClickedEndTriggerButton` to create a handler for it.

`kQTEventMouseEnter`

Event sent to a sprite when the mouse first enters it. This sprite becomes the one that will receive the `kQTEventMouseExit` event. This event is sent whether or not the mouse button is currently pressed. If two or more sprites overlap, the sprite in front receives the event. You set the ID of a `kQTEventType` atom to `kQTEventMouseEnter` to create a handler for it.

`kQTEventMouseExit`

Event sent to the sprite that received the last `kQTEventMouseEnter` event when the mouse is either no longer over it, or enters another sprite which is in front of it. This event is sent whether or not the mouse button is

currently pressed. You set the ID of a `kQTEventType` atom to `kQTEventMouseExit` to create a handler for it.

`kQTEventIdle`

This event is sent to each sprite in a sprite track only if the sprite track has the sprite track property `kSpriteTrackPropertyQTIdleEventsFrequency` set to a value other than the default value which is `kNoQTIdleEvents`. You set the ID of a `kQTEventType` atom to `kQTEventIdle` to create a handler for it.

`kQTEventFrameLoaded`

This event is sent to the sprite track when the current sprite track frame is loaded and contains a handler for this event. This event is not sent to a particular sprite; only one handler of this type may be present in a single sprite track frame. A typical use of this event would be to initialize sprite track variables.

Note that each frame can have its own handler. If the frame is a key frame, the scope of the handler is from the key frame until the next key frame unless the handler is overridden. If an override containing a handler for a particular event is followed by another override with no handler for that event, the key frame's handler for that event is once again used.

To create a handler for this type of event, you add an atom of type `kQTEventTypeFrameLoaded` to the sample as a sibling of the `kSpriteAtomType` atom with an ID of 1.

Target Constants

Each action is performed upon a particular type of target, usually an element of the movie such as a sprite or a track. The track that is handling a QuickTime event is the default track target. If the event is being handled by a sprite, then that sprite is the default sprite target. For example, if a sprite has a handler for a mouse down event, when it receives that event it becomes the default sprite and its sprite track becomes the default sprite track while the QuickTime event is being handled.

To specify a target, you add a `kActionTarget` atom, and then add one or more of the target type atoms as children. Tracks and sprites may be specified by name,

ID, or index. Tracks may additionally be specified by type and index within the type.

Note

For sprite targets, you may specify a particular sprite track as well as a particular sprite in that track. ♦

Constant descriptions

<code>kActionTarget</code>	You add an atom of this type as a child of a <code>kActionAtom</code> , if you wish to specify a target other than the default target for a given action. This atom should contain one or more target-related child atoms.
<code>kTargetMovie</code>	You add this atom to specify the movie as the action's target. Since the movie is the default target for movie actions, this atom is optional.
<code>kTargetTrackName</code>	You add an atom of this type to specify a track by name as the action's track target. The leaf data is a Pascal string containing the name of the track. Name matching is not case-sensitive.
<code>kTargetTrackType</code>	You add an atom of this type to specify a track by type as the action's track target. The leaf data is an <code>OSType</code> containing the type of the Track. To specify a particular index within this Track type, you may additionally add a <code>kTargetTrackIndex</code> atom. If no <code>kTargetTrackIndex</code> is present, an index of one will be used as a default.
<code>kTargetTrackIndex</code>	You add an atom of this type to specify a track by index as the action's track target. The leaf data is a <code>long</code> containing the index of the track. Unless a <code>kTargetTrackType</code> atom is present, the index refers to an index within all track types in the movie. By adding a <code>kTargetTrackType</code> atom, you may specify an index within a particular track type.
<code>kTargetTrackID</code>	You add an atom of this type to specify a Track by ID as the action's track target. The leaf data is a <code>long</code> containing the ID of the track.
<code>kTargetSpriteName</code>	You add an atom of this type to specify a sprite by name as the action's sprite target. The leaf data is a Pascal String containing the name of the sprite. You may additionally add Track-related target atoms to specify that the sprite is

	in a sprite track other than the default sprite track. Name matching is not case sensitive
<code>kTargetSpriteIndex</code>	You add an atom of this type to specify a sprite by index as the action's sprite target. The leaf data is a <code>short</code> containing the index of the sprite. You may additionally add track-related target atoms to specify that the sprite is in a sprite track other than the default sprite track.
<code>kTargetSpriteID</code>	You add an atom of this type to specify a sprite by ID as the action's sprite target. The leaf data is a <code>QTAtomID</code> containing the ID of the sprite. You may additionally add track-related target atoms to specify that the sprite is in a sprite track other than the default sprite track.

Action Constants

Each individual action in an action list is contained by an atom of type `kActionAtom`. Its child atoms define which action it is, what target in the movie it operates on, what the values of each of its parameters are, and any parameter options for each parameter.

Since each action is intended to be sent to a particular type of target, the action constants are explained below, grouped by their target type.

Note

If minimum and maximum values are not specified, then they may be assumed to be the full range of the parameter's data type. ♦

Constant descriptions

<code>kAction</code>	This atom is a container for a single sprite action. Add an atom of this type for each action in an action list. The list of actions will be executed in order, based on the index of these atoms, from one to the number of <code>kAction</code> atoms present. The IDs may be any unique IDs. This atom's parent type is commonly <code>kQTEventType</code> . It may also be a child atom of a <code>kQTEventFrameLoaded</code> atom, and of a <code>kActionListAtomType</code> atom for nested action lists.
<code>kWhichAction</code>	

This atom specifies the type of action that will be executed. Add a child atom of this type to each `kAction` atom, setting its ID to 1. The atom's leaf data is a `long` which contains the constant describing which action to execute. For example, to create a set movie volume action, set the leaf data of the `kWhichAction` atom to `kActionMovieSetVolume`.

`kCommentAtomType`

This atom allows you to comment an action or an event in a language independent manner. Tools which decompile action lists may choose to display these comments. You may add any number of comments to each action or event. The order is defined by the atom index.

The parent atom type of the `kCommentAtomType` is either `kAction`, `kQTEventType`, or `kQTEventFrameLoaded`. The leaf data is a C string.

Movie Action Constants

The following movie action constants enable you to set the properties of a movie. For example, you can use these constants to create play and pause buttons which control the movie's rate.

Constant descriptions

`kActionMovieSetVolume`

Supported Flags: `IsDelta`, `WrapsAround`

Param1: [short volume]

Default Min: 0, **Default Max:** 255

This sets the movie's volume level.

`kActionMovieSetRate`

Supported Flags: `IsDelta`, `WrapsAround`

Param1: [Fixed rate]

Default Min: `minFixed`, **Default Max:** `maxFixed`

This sets the movie's playback rate. A rate of 1 means play back at normal speed. A rate of two means play back at double speed. A rate of 0 means stop. Negative rates make the movie play backwards. Rates may be fractional.

`kActionMovieSetLoopingFlags`

Supported Flags: none

Param1: [long loopingFlags]

This sets the looping mode of movie playback. Zero means no looping. Setting the `loopTimeBase` flag means that the movie will loop; additionally setting the `palindromeLoopTimeBase` flag causes the movie to loop in palindrome fashion, meaning that once it reaches the end, it goes backwards until reaching the beginning, at which point it will go forward again. Note that the flags `loopTimeBase` and `palindromeLoopTimeBase` are OR-combined together.

`kActionMovieGoToTime`

Supported Flags: none

Param1: [TimeValue time]

This sets the movie's current time. This value is expressed in the movie's timescale.

`kActionMovieGoToTimeByName`

Supported Flags: none

Param1: [Str255 timeName]

Sets the movie's current time to the one in the movie corresponding to the chapter named `timeName`.

`kActionMovieGoToBeginning`

Supported Flags: none

No Params

Sets the time to the beginning of the movie.

`kActionMovieGoToEnd`

Supported Flags: none

No Params

Sets the time to the end of the movie.

`kActionMovieStepForward`

Supported Flags: none

No Params

This causes the movie to step forward in the same fashion as pressing the step forward button in the movie controller.

`kActionMovieStepBackward`

Supported Flags: none

No Params

This causes the Movie to step backward in the same fashion as pressing the step backward button in the movie controller.

Wired Movies API Reference

`kActionMovieSetSelection`**Supported Flags:** none

Param1: [TimeValue startTime]

Param2: [TimeValue endTime]

Sets the movie's selection to be the time range specified by the startTime and endTime time values.`kActionMovieSetSelectionByName`**Supported Flags:** none

Param1: [Str255 startTimeName]

Param2: [Str255 endTimeName]

Sets the movie's selection to be the time range specified by the startTimeName and endTimeName chapter names.`kActionMoviePlaySelection`**Supported Flags:** IsToggle

Param1: [Boolean selectionOnly]

When set to true, the movie plays only the current movie selection. The movie selection may be set using`kActionMovieSetSelection` **or**`kActionMovieSetSelectionByName`.`kActionMovieSetLanguage`**Supported Flags:** none

Param1: [long language]

Sets the movie's current language. This causes the tracks associated with that language to be enabled and track's associated with other languages to be disabled.

Actions for All Tracks

This action for all tracks enables or disables a track. You can use it, for example, to switch between two different sound tracks, or two different tween tracks.

Constant descriptions

`kActionTrackSetEnabled`**Supported Flags:** IsToggle

Param1: [Boolean enabled]

Enables or disables a track.

New Wired Movie Actions and Operands in QuickTime 4

This section describes the new movie actions and operands available to your application in QuickTime 4.

New Actions

QuickTime 4 provides support for the following new wired movie features:

- wired movie cursors (page 106). There are a few built-in cursors, but you can add your own cursors as well.
- runtime sprites (page 111). Runtime sprites may be dynamically created and disposed using new sprite media handler routines or by using the new wired actions.

Actions for All Tracks

Constant descriptions

`kActionTrackSetCursor`

Supported Flags: none

Parm1: [QTAtomID cursorID]

This sets the cursor to the one specified by `cursorID`. Note that the target is a track. This is because tracks may supply their own custom cursors.

You set the cursor ID as follows:

`cursorID = 0`

Set the cursor to the system default cursor.

`cursorID (1..1000)`

You set the cursor to a built-in QuickTime cursor. You use values from this enumeration in `Movies.h`.

Wired Movies API Reference

```
enum {
    kQTCursorOpenHand          = 128,
    kQTCursorClosedHand       = 129,
    kQTCursorPointingHand     = 130,
    kQTCursorRightArrow       = 131,
    kQTCursorLeftArrow        = 132,
    kQTCursorDownArrow        = 133,
    kQTCursorUpArrow          = 134
};
```

```
cursorID > 1000
```

You set the cursor to one included in the movie. Specifically, it is included in the target track's `MediaPropertiesAtom` `atomdata`. This is an atom of type 'crsr' with an atom ID corresponding to the desired cursor ID (greater than 1000). The atom's data is simply a Macintosh 'crsr' resource. On Windows, a black-and-white version is currently used.

```
kActionTrackSetGraphicsMode
```

Supported Flags: none

Parm1: [ModifierTrackGraphicsModeRecord graphicsMode]

Sets a spatial track's graphics mode to the one specified.

```
kActionSendQTEventToTrackObject
```

Supported Flags: none

Parm1: [[[TrackObjectTargetAtoms]] trackTarget]

Parm2: [QTEventRecord theEvent]

Similar to `kActionSendQTEventToSprite`, but applies to any wired media handler that contains track objects such as Sprite, Text, and QuickDraw 3D. This lets you send standard or custom events to track objects.

Actions That Do Not Have a Target

Constant descriptions

```
kActionStatusString
```

Supported Flags: none

```

Parm1: [CString statusString]
Param2: [flags]

```

You may use this action to instruct the QuickTime plug-in to display a URL link or other information in the status area of the browser. In order to do this, you set the flags parameter to `kStatusStringIsURLLink` and the `statusString` to a C string containing the URL or other information to display.

QuickTime Streaming uses the status string action to report some types of streaming status. Your application may listen to this status by installing a movie controller filter procedure. In this case, the `kStatusStringIsStreamingStatus` bit will be set in the flags field. Additionally, if the `kStatusHasErrorCode` bit is set, then the high 16 bits of `stringTypeFlags` is an error code number.

```

enum {
    kStatusStringIsURLLink          = 1L << 1,
    kStatusStringIsStreamingStatus = 1L << 2,
    kStatusHasErrorCode             = 1L << 3
};

```

`kActionAddChannelSubscription`

Supported Flags: none

```

Param1: [CString titleString]
Param2: [CString URL]
Param3: [CString pictureURL] (optional)

```

Adds a channel guide subscription of `kBookmarkSubscriptionType` with the supplied title, URL, and, optionally, a picture URL to the QuickTime Preferences file.

`kActionRemoveChannelSubscription`

Supported Flags: none

```

Param1: [CString URL]

```

Removes the channel guide subscription of

`kBookmarkSubscriptionType` from the QuickTime Preferences file if it exists.

`kActionOpenCustomActionHandler`

Supported Flags: none

Param1: [long handlerID]

Param2: [ComponentDescription desc]

This action attempts to locate and open a custom action-handling component with the supplied component description and assign it the specified handlerID. If successful, the handler is kept open so that its state is retained between calls. It will be closed automatically when the movie is disposed.

This call may fail if there is no component of the specified type available, or if there is already a custom handler open with the same handlerID. You may use the `kOperandUniqueCustomActionHandlerID` to obtain a unique id for a handler you wish to open. You may use the `kOperandCustomActionHandlerIDIsOpen` to determine if a handler with a certain handlerID is open.

`kActionApplicationNumberAndString`

Supported Flags: none

Param1: [long aNumber]

Param2: [Str255 aString]

This action does nothing. Applications may watch for this action and know that the first parameter is a number and second parameter a string. It could be used to communicate information from a movie to an application. To do this, your application needs to hook the movie controller's `mcActionExecuteOneActionForQTEvent` action and examine the atom container to determine if the current action is `kActionApplicationNumberAndString`, and if so, look at the parameter atoms.

Another way to pass a string to an application would be to use `kActionStatusString` with a custom flag set. To do this, an application only needs to hook the movie controller's `mcActionShowStatusString` action.

Actions for Sprite Tracks

A sprite created at runtime is associated with the current sprite key frame. It will remain alive until a new sprite key frame is loaded. It has all the regular properties of a sprite including the new sprite property `kSpritePropertyActionHandlingSpriteID` which delegates its handling of wired events to another sprite. In QuickTime 4, runtime sprites cannot have their own unique event handling atoms.

Constant descriptions

`kActionSpriteTrackNewSprite`

Supported flags: none

Param1: [QTAtomID spriteID]

Param2: [short imageIndex]

Param3: [MatrixRecord matrix]

Param4: [short visible]

Param5: [short layer]

Param6: [ModifierTrackGraphicsModeRecord
graphicsMode]

Param7: [QTAtomID actionHandlingSpriteID]

Dynamically creates a new sprite with the properties specified by the parameters. You may pass zero for the `spriteID` and a unique ID will be assigned.

`kActionSpriteTrackDisposeSprite`

Supported flags: none

Param1: [QTAtomID spriteID]

Disposes the sprite with the specified ID. Note that this disposes the sprite object in memory, it does not dispose the sprite from the media. If you dispose a sprite that exists in a sprite track's media, it will be created again the next time the key frame sample is loaded.

`kActionSpriteTrackSetVariableToString`

Supported flags: none

Param1: [QTAtomID variableID]

Param2: [Cstring theString]

Sets the value of the sprite track variable specified by `variableID` to `theString`. This replaces the previous value of

the variable. Variables may be either a string or a floating-point number.

`kActionSpriteTrackConcatVariables`

Supported flags: none

Param1: [QTAtomID firstVariableID]

Param2: [QTAtomID secondVariableID]

Param3: [QTAtomID resultVariableID]

Concatenates the string in the sprite track variable specified by `secondVariableID` to the end of the string in `firstVariableID` and places the result in the variable `resultVariableID`. Uninitialized string variables are the empty string. Variables containing floating-point numbers are coerced to strings.

`kActionSpriteTrackSetVariableToMovieURL`

Supported flags: none

Param1: [QTAtomID variableID]

Param2: <[MovieTargetAtoms externalMovie]>

Sets the value of a sprite track variable to a string containing the URL of the movie that contains the sprite track, or optionally the URL of an external movie if the second parameter is used.

Actions for QD3D Named Objects

The new action target constant `kTargetQD3DNamedObjectName` may be used to specify a named object within a QuickDraw 3D track. The leaf data of this atom is a `CString`.

Constant descriptions

`kActionQD3DNamedObjectTranslateTo`

Supported flags: none

Param1: [Fixed x]

Param2: [Fixed y]

Param3: [Fixed z]

Translates the named object to (x, y, z).

kActionQD3DNamedObjectScaleTo

Supported flags: none

Param1: [Fixed xScale]

Param2: [Fixed yScale]

Param3: [Fixed zScale]

Scales the named object by (xScale, yScale, zScale).

kActionQD3DNamedObjectRotateTo

Supported flags: none

Param1: [Fixed xDegrees]

Param2: [Fixed yDegrees]

Param3: [Fixed zDegrees]

Rotates the named object to (xDegrees, yDegrees, zDegrees).

Actions for Flash Tracks

For more information about Flash track features and support which are new to QuickTime 4, refer to Chapter 5, “Flash Media Handler.”

Constant descriptions

kActionFlashTrackSetPan

Supported flags: none

Param1: [short xPercent]

Param2: [short yPercent]

Pans a Flash Track by the specified x and y percentages.

kActionFlashTrackSetZoom

Supported flags: none

Param1: [short zoomFactor]

Zooms a Flash track by zoomFactor.

kActionFlashTrackSetZoomRect

Supported flags: none

Param1: [long left]

Param2: [long top]

Param3: [long right]

Param4: [long bottom]

Zooms a Flash track to the specified rectangle.

`kActionFlashTrackGotoFrameNumber`

Supported flags: none

Param1: [long frameNumber]

Sets the movie time of the movie containing the Flash track to the time that corresponds to the specified flash frame number.

`kActionFlashTrackGotoFrameLabel`

Supported flags: none

Param1: [CString frameLabel]

Sets the movie time of the movie containing the Flash track to the time that corresponds to the flash frame with label equal to frameLabel.

New Operands

These new operands in QuickTime 4 supply information about the environment that the movie is running in.

Constant descriptions

`kOperandSubscribedToChannel`

Param1: [CString channelURL]

Returns true if the QuickTime Preference file contains a subscription to the channel specified by channelURL; otherwise returns false.

`kOperandUniqueCustomActionHandlerID`

Returns an unused custom action handler ID. This value may be used with `kActionOpenCustomActionHandler` and stored in a sprite track variable.

`kOperandCustomActionHandlerIDIsOpen`

Param1: [long handlerID]

Returns true if a custom action handler with the specified ID is currently open.

`kOperandConnectionSpeed`

Returns the connection speed setting in a users QuickTime preferences file.

`kOperandGMTDay`

Returns the day value (1..31) for the current Greenwich Mean time.

`kOperandGMTMonth`

Returns the month value (1..12) for the current Greenwich Mean Tim time.

`kOperandGMTYear`

Returns the year value for the current Greenwich Mean time.

`kOperandGMTHours`

Returns the hours value in 24 hour time (0..23) for the current Greenwich Mean time.

`kOperandGMTMinutes`

Returns the minutes value (0..59) for the current Greenwich Mean time.

`kOperandGMTSeconds`

Returns the seconds value (0..59) for the current Greenwich Mean time.

`kOperandLocalDay`

Returns the day value (1..31) for the current local time.

`kOperandLocalMonth`

Returns the month value (1..12) for the current local time.

`kOperandLocalYear`

Returns the year value for the current local time.

`kOperandLocalHours`

Returns the hours value in 24 hour time (0..23) for the current local time.

`kOperandLocalMinutes`

Returns the minutes value (0..59) for the current local time.

`kOperandLocalSeconds`

Returns the seconds value (0..59) for the current local time.

`kOperandRegisteredForQuickTimePro`

Returns true if the user is registered for the Pro version of QuickTime, otherwise returns false.

`kOperandPlatformRunningOn`

Returns 1 if the computer platform is Macintosh, 2 if it is Windows.

`kOperandQuickTimeVersion`

Returns the version of QuickTime that is in use.

`kOperandComponentVersion`

Param1: [CString type]
 Param1: [CString subType]
 Param1: [CString manufacturer]

Returns the version of a specific QuickTime component. The component is specified using four character string codes for the type, subType, and manufacturer. If the component is not available, a version number of zero is returned.

`kOperandOriginalHandlerRefcon`

Returns the refcon of the original event handler. This is useful if you are using multiple sprites that delegate the handling of actions to a common handler using the new `actionHandlerID` property. The original handler's refcon in this case is the ID of the sprite that was originally clicked on or sent a custom event.

Actions for Spatial Tracks

The following actions for spatial tracks let you change the spatial properties in movies, such as scaling, resizing, or changing shapes.

Constant descriptions

`kActionTrackSetMatrix`

Supported Flags: `IsDelta`, `WrapsAround`

Param1: `[MatrixRecord matrix]`

For each matrix element, the default maximum and minimum values are the largest and smallest possible Fixed values.

Sets the target track's matrix, allowing you to move, resize, rotate, and otherwise distort a track's shape. If you set the Track's matrix to a value that makes its display bounds fall outside of the movie's bounds, then the movie's bounds are resized to accommodate the new track box. Although this work's fine in the QuickTime Movie Player, be warned that not all applications support this type of dynamic movie resizing cleanly.

`kActionTrackSetLayer`

Supported Flags: `IsDelta`, `WrapsAround`

Param1: `[short layer]`

Sets the target track's layer number. This is used to specify its front-to-back order relative to the other spatial track's in the movie. The smaller the layer number, the more forward the track appears.

`kActionTrackSetClip`

Supported Flags: `none`

Param1: `[RgnHandle clip]`

Sets the track's clipping region. This parameter contains QuickDraw Region data.

Actions for Sound Tracks

You use these actions for sound tracks to set volume and balance. With these, you can create a button or slider to control sound tracks.

Constant descriptions`kActionTrackSetVolume`**Supported Flags:** `IsDelta`, `WrapsAround`**Param1:** [short volume] **Default Min:** 0, **Default Max:** 255

Sets the track's volume level.

`kActionTrackSetBalance`**Supported Flags:** `IsDelta`, `WrapsAround`**Param1:** [short volume] **Default Min:** -128, **Default Max:** 128

Sets the track's left to right balance.

The range of numbers is as follows: -128 = only left;
128 = only right; 0 = even amounts of left and right.

Actions for Sprites in a Sprite Track

The following actions enable you to control a sprite's spatial properties.

Constant descriptions`kActionSpriteSetMatrix`**Supported Flags:** `IsDelta`, `WrapsAround`**Param1:** [MatrixRecord matrix]

Sets the target sprite's matrix, allowing you to move, resize, rotate, and otherwise distort a sprite's shape. Sprites are clipped by their containing sprite track's bounds and clip region.

`kActionSpriteSetImageIndex`**Supported Flags:** `IsDelta`, `WrapsAround`**Param1:** [short imageIndex] **Default Min:** 1, **Default Max:** num imagesSets the target sprite's image index. Each sprite track `keyFrame` contains a list of images. Setting a sprite's image index selects which image from this list is currently displayed. The image index ranges from 1 to the number of images.`kActionSpriteSetVisible`**Supported Flags:** `isToggle`**Param1:** [short visible]

Shows or hides the target Sprite.

`kActionSpriteSetLayer`

Supported Flags: `IsDelta`, `WrapsAround`

Param1: `[short layer]`

Sets the target sprite's layer number. This is used to specify its front-to-back order relative to the other sprites in the sprite track. The smaller the layer number, the more forward the sprite appears. Note that 32767 indicates that this is a background sprite.

`kActionSpriteSetGraphicsMode`

Supported Flags: `IsDelta`

Param1: `[ModifierTrackGraphicsModeRecord graphicsMode]`

Sets the target Sprite's graphics mode.

`kActionSpritePassMouseToCodec`

Supported Flags: `none`

No Params

Passes the location of the mouse event to the codec that is drawing the sprite's current image. (Note that currently, only the Ripple codec accepts clicks, causing a ripple effect originating from the location).

It only makes sense to use this action in response to mouse-related events.

`kActionSpriteClickOnCodec`

Supported Flags: `none`

Param1: `[Point localLoc]`

Passes the point `localLoc` to the codec that is drawing the sprite's current image. (Note that currently, only the Ripple codec accepts clicks, causing a ripple effect originating from the location.)

This action is similar to `kActionPassMouseToCodec` except the location to click on is a parameter, so it may be used in response to any type of event.

`kActionSpriteTranslate`

Supported Flags: `none`

Param1: `[Fixed x]`

Param2: `[Fixed y]`

Param3: `[Boolean isAbsolute]`

If the `isAbsolute` parameter is `true`, this moves the sprite to the absolute location specified by the `x` and `y` parameters; if

the `isAbsolute` parameter is `false`, it specifies how far from the current location to move the sprite. The coordinate system for `x` and `y` is the sprite track's source space.

`kActionSpriteScale`

Supported Flags: none

Param1: [Fixed `xScale`]

Param2: [Fixed `yScale`]

Scales the target sprite by `xScale` and `yScale` about its current image's registration point. For example, to double a sprite's width and half its height, you would set `xScale` to two and `yScale` to one-half.

`kActionSpriteRotate`

Supported Flags: none

Param1: [Fixed degrees]

Rotates the target sprite about its current image's registration point. The amount of rotation is specified by degrees.

`kActionSpriteStretch`

Supported Flags: none

Param1: [Fixed `p1x`]

Param2: [Fixed `p1y`]

Param3: [Fixed `p2x`]

Param4: [Fixed `p2y`]

Param5: [Fixed `p3x`]

Param6: [Fixed `p3y`]

Param7: [Fixed `p4x`]

Param8: [Fixed `p4y`]

The eight parameters specify four corners of a four-sided polygon into which the sprite's image is "stretched." The coordinate system for points is the sprite track's source space.

Actions for QuickTime VR Tracks

Constant descriptions

`kActionQTVRSetPanAngle`

Supported Flags: `IsDelta`, `WrapsAround`

Param1: [float `panAngle`]

Default Min: min Pan Allowed By Media,
 Default Max: max Pan Allowed By Media
 Sets the target QuickTime VR track's pan angle.

`kActionQTVRSetTiltAngle`

Supported Flags: `IsDelta`, `WrapsAround`
 Param1: [float tiltAngle]
 Default Min: min Tilt Allowed By Media,
 Default Max: max Tilt Allowed By Media
 Sets the target QuickTime VR track's tilt angle.

`kActionQTVRSetFieldOfView`

Supported Flags: `IsDelta`, `WrapsAround`
 Param1: [float fieldOfView]
 Default Min: min FieldOfView Allowed By Media,
 Default Max: max FieldOfView Allowed By Media
 Sets the target QuickTime VR track's field of view.

`kActionQTVRShowDefaultView`

Supported Flags: none
 No Params
 Causes the target QuickTime VR track to show its default view.

`kActionQTVRGoToNodeID`

Supported Flags: none
 Param1: [UInt32 nodeID]
 Causes the multi-node target QuickTime VR track to go to the specified node ID.

Actions for Music Tracks

The following describes the constants you can use as actions for music tracks. For more detailed explanations of how parameters for pitch and velocity are interpreted, see *QuickTime Music Architecture*.

Constant descriptions

`kActionMusicPlayNote`

Supported Flags: none
 Param1: [long sampleDescIndex]
 Param2: [long partNumber]
 Param3: [long delay]

```
Param4: [long pitch]
Param5: [long velocity]
Param6: [long duration]
```

This causes the target music track to play a note. Since the current selection of instruments for a music track is determined by its sample description, you use the `sampleDescIndex` parameter to select which sample description to use. The `partNumber` parameter specifies which part to use within the sample descriptions list. The default is 1.

If you want the note to be delayed, you may pass a positive value for the delay parameter, which is interpreted in the time scale of the music track. Pass 0 for no delay. The pitch parameter selects which note to play; for example, Middle C is 60, Middle B is 59. The velocity specifies the volume that the note is played at. The duration specifies the length of time that the note is played for and is interpreted in the time scale of the music track.

When using a music track to play notes, you should set the `kMusicFlagDontSlaveToMovie` flag in the `MusicDescription`'s `musicFlags` field. The music track used for playing notes should not contain note information. If you wish to layer notes on top of other music, you should use two separate music tracks.

By setting duration to `kNoteEventDurationMax`, the note will continue playing until you send another `kActionMusicPlayNote` for the same pitch with a velocity of 0. This allows you to hold a note for an interactive period of time. For example, on `mouseDown`, you play a note, and on `mouseUp` to turn it off.

```
kActionMusicSetController
```

Supported Flags: none

```
Param1: [long sampleDescIndex]
Param2: [long partNumber]
Param3: [long delay]
Param4: [long controller]
Param5: [long value]
```

Sets a controller value for a part in the target music track. Since the current selection of instruments for a music track

is determined by its sample description, you use the `sampleDescIndex` parameter to select which sample description to use. The `partNumber` parameter specifies which part to use within the sample descriptions list.

If you want the controller change to be delayed, you may pass a positive value for the delay parameter, which is interpreted in the time scale of the music track. Pass 0 for no delay. Controller values control things, such as pitch bend and reverb. For more information about available controllers, see *QuickTime Music Architecture*.

Actions for Sprite Tracks

Constant descriptions

`kActionSpriteTrackSetVariable`

Supported Flags:none

Param1: [QTAtomID variableID]

Param2: [float value]

Sets the value of the variable specified by `variableID` for the target sprite track to value.

Each sprite track in a movie has its own set of variables. Variables are runtime only; they are not saved and restored. If you need to initialize variables to certain values, you can use the `kActionSpriteTrackSetVariable` action in response to the `kQTEventFrameLoaded` event.

Actions That Do Not Have a Target

The following perform global actions to a movie.

Constant descriptions

`kActionGoToURL`

Supported Flags:none

Param1: [C string]

If the movie is currently being presented using the Web browser plug-in, this causes the browser to go to the specified URL. If the movie is being presented by

MoviePlayer, a Web browser is launched and goes to the specified URL. You may optionally specify a particular frame within a Web page.

To specify a URL, use the standard Web address format—for example, `http://www.apple.com`. You may optionally use angle brackets—for example, `<http://www.apple.com>`. To specify a particular frame within a URL, you must use angle brackets followed by `T<frameName>`—for example, `<http://www.apple.com>T<frameName>`.

`kActionSendQTEventToSprite`

Supported Flags: none

Param1: [(SpriteTargetAtoms)]

Param2: [QTEventRecord theEvent]

Sends `theEvent` to the sprite specified by

`SpriteTargetAtoms`. You may send any event to a sprite, including a custom event that you define. Note that `kActionSendQTEventToSprite` has no target, since it is handled by the system, although you do specify a target `Param1` for the event that it sends `Param2`.

When sending a custom event, make sure the events constant does not conflict with an existing event. If you need to share a complicated event handler among many sprites, you may wish to define a custom event handler on a single sprite and have the others send it the custom event using `kActionSendQTEventToSprite`.

`kActionApplicationNumberAndString`

Supported Flags: none

Param1: [long aNumber]

Param2: [Str255 aString]

QuickTime does nothing when this action is executed; it is intended to be used by applications that wish to “hook” it, using the movie controller’s `MCSetActionFilter` routine with the new constant `mcActionExecuteOneActionForQTEvent`. Applications may look at the number and string parameters to determine what they want to do when the action is executed.

`kActionDebugStr`**Supported Flags:**none

Param1: [Str255 theString]

Passes theString **through the movie controller's filter proc using the** mcActionShowMessageString **constant.**

Applications that wish to display the message string may use the movie controller's MCSetActionFilter **routine along with the new constant** mcActionShowMessageString **in order to receive the string. QuickTime does not display the string.**

Your application can intercept actions using the movie controller's filter procedures. This action serves as a placeholder, so that your application can look for actions of this type with an aNumber **or** aString **parameter that identifies the application. QuickTime will not do anything in response to this action. Your application can then do anything it wants in response to the action. If your application is an authoring tool, for example, you could set the values of the** aNumber **and** aString **parameters. This is currently the only action that is specific to applications.**

`kActionPushCurrentTime`**Supported Flags:**none**No Params**

Places the current movie time onto the top of the movie controller's stack of times.

The movie controller maintains a stack data structure of movie times. The stack is manipulated using these actions.

`kActionPushCurrentTimeWithLabel`**Supported Flags:**none

Param1: [Str255 theLabel]]

Pushes the current time onto the top of the movie controller's stack of times, along with a label. This label is not a chapter name, it is a tag that can be used in conjunction with kActionPopAndGotoLabeledTime.

`kActionPopAndGotoTopTime`**Supported Flags:**none**No Params**

Retrieves the top time from the movie controller's stack of times and removes it from the stack. The movie's current time is then set to the time retrieved.

`kActionPopAndGotoLabeledTime`

Supported Flags: none

Param1: [Str255 theLabel]

Searches from the top of the movie controller's stack of times towards the bottom until it finds a time with the specified label. If found, it removes all times from the top of the stack through the labeled time and sets the movie's current time to this time. Note that only the topmost time with a matching label is popped, not all occurrences times with the same label.

Control Statement Actions

Control statement actions have no target. They are used to control the flow of execution in an action list, and provide a means to create nested action lists. Using these actions along with the state provided by track variables allows you to create more sophisticated event handlers.

Constant descriptions

`kActionCode`

Supported Flags: none

Param1: [(CaseStatementActionAtoms)]

Provides a conditional control structure. The `CaseStatementActionAtoms` allow for pairs of expressions and actions to be defined. The list of expressions is evaluated, when an expression evaluates to `true`, no other expressions are evaluated and the list of actions associated with that expression is executed.

Nested control structures are possible, since they are themselves actions.

`kActionWhile`

Supported Flags: none

Param1: [(WhileStatementActionAtoms)]

Provides a looping control structure. The `WhileStatementActionAtoms` define an expression and a list of actions to be defined. While the expression still evaluates to `true`, the list of actions is performed.

If your list of actions does not contain an action which will eventually cause the expression to evaluate to `false`, the control structure will be caught in an infinite loop. Nested

control structures are possible, since they are themselves actions.

`kConditionalAtomType`

This atom is used in conjunction with the case and while statement actions. It contains atoms which describe a Boolean expression and an action list, which is executed on the condition that the Boolean expression evaluates to true. In both cases, this atom is a child atom of the action's only `kActionParameter` atom. The atom contains two child atoms, one of type `kExpressionContainerAtomType` and one of type `kActionListAtomType`.

`kActionListAtomType`

This atom is used to contain an action list to be performed by the `kActionCode` and `kActionWhile` actions when an associated expression evaluates to true. In both cases, the atom's parent is of type `kConditionalAtomType`.

Action Parameter Constants

`kActionParameter`

This atom describes one parameter for a given action. Add one atom of this type, as a child of the `kAction` atom, for each parameter that the action requires. The number of parameters and their data types is described for each action in the section “Action Constants” (page 103).

The atom's index, not ID, correspond to the action's parameter numbers. The atom commonly contains leaf data with the same type as its parameter's data type. If the data type is numeric, then it may be described by an expression, in which case it would have a child atom of type `kExpressionContainerAtomType`. For the case and while statement actions the `kActionParameter` always has a `kConditionalAtomType` child atom.

`kActionFlags`

This optional atom may be used to specify flags which modify a parameter's value. The leaf data of this atom is a `long` which contains the flags. The ID of this atom must be set to the ID of the corresponding `kActionParameter` atom which it modifies. This atom's parent is a `kAction` atom.

The following flags may be used. Note that the action flags work in conjunction with the values specified by the

`kActionParameterMinValue` and `kActionParameterMaxValue` atoms.

`kActionFlagActionIsDelta`

The parameter's value will be added to the current value instead of replacing the current value. This is useful, for example, if you want an action to increment a Movie's volume by a fixed amount. The new value will be pinned to the minimum and maximum values. Use the

`kActionParamMinValue` and `kActionParamMaxValue` atoms to set the minimum and maximum values. You set this flag in the leaf data of a `kActionFlags` atom.

When this flag is set, the parameter value is interpreted as a signed value, allowing the delta to be positive or negative. This is true even if the parameter itself is normally an unsigned value, such as an entry of a graphic mode's `RGBColor`.

`kActionFlagParameterWrapsAround`

If after applying the parameter to the current value the new value is greater than the maximum value or less than the minimum value, then the value wraps around to a value which is in range.

This is useful, for example, if you want to create an action to cycle between sprite image indices, or layers. Use the `kActionParamMinValue` and `kActionParamMaxValue` atoms to set the minimum and maximum values. You set this flag in the leaf data of a `kActionFlags` atom.

`kActionFlagActionIsToggle`

This flag is supported by actions with a single two state parameter. If this flag is set, the parameter value is ignored, and instead the current value is toggled to its other state. This is useful, for example, if you want a single action to toggle a sprite's visibility between visible and invisible. You set this flag in the leaf data of a `kActionFlags` atom.

`kActionParameterMinValue`

You add an atom of this type for each parameter which you wish to enforce a minimum value upon. Action parameters have default minimum values, so you only need to add this atom if you wish to override this default value. The atom ID should match the ID of the associated `kActionParameter`

atom. Note that not all actions support this optional atom. This atom's parent is a `kAction` atom. The size of the leaf data depends on the parameter's data type.

`kActionParameterMaxValue`

Add an atom of this type for each parameter which you wish to enforce a maximum value upon. Action parameters have default maximum values so you only need to add this atom if you wish to override this default value. The atom ID should match the ID of the associated `kActionParameter` atom. Note that not all actions support this optional atom. This atom's parent is a `kAction` atom. The size of the leaf data depends on the parameter's data type.

For `MatrixRecord` parameters, the delta is concatenated with the current matrix. Each of the nine matrix elements are constrained (pinned or wrapped around their max and min) separately after the concatenation. This differs from scalar values in that overflow from the delta operation is not detected, so it is actually possible to have some elements wrap around even if the wrap around flag is not set. This can easily be avoided by choosing values for the matrix param, and min and max matrices.

Each matrix contains nine elements, and each element is concatenated separately. If you have matrices for specifying maximum and minimum values, you define maximum and minimum values for each of the individual matrix elements. If you do not provide matrices for specifying maximum and minimum values, the value of each element can wrap around if arithmetic overflow occurs. However, there is no checking for overflow when performing a delta operation on a matrix; this is in contrast to scalar values, for which overflow can always be checked.

For `ModifierTrackGraphicsModeRecord` parameters, the delta's `RGBColor` elements are each added to the current `ModifierTrackGraphicsModeRecord`'s `RGBColor` elements. Each red, green, and blue element is constrained separately. As with `MatrixRecord` records, you can have `ModifierTrackGraphicsModeRecord` records that specify maximum and minimum values for each field of the record.

Expression-Related Constants

Expressions may be used for numeric parameter types, targets, and with the control statement actions.

Constant descriptions

`kExpressionContainerAtomType`

This atom is used to contain an expression. Expressions are used to describe numeric paramers, sprite and track IDs and indices, Boolean expressions for the case and while statements, and as a special operand type which allows for the nested sub-expressions within expressions. The complete grammar of expressions is described in Appendix B.

The `kExpressionContainerAtomType` has either a `kOperatorAtomType` or a `kOperandAtomType` atom as its only child atom. The parent type of the `kExpressionContainerAtomType` atom may be any `kActionParameter` atom associated with a numeric data type, or any of the following: `kTargetTrackIndex`, `kTargetTrackID`, `kTargetSpriteIndex`, `kTargetSpriteID`, `kConditionalAtomType`, or `kExpressionOperandAtomType`.

`kOperatorAtomType`

This atom is used to describe an operation, such as multiplication or addition, which forms part of an expression. The atom's ID defines which operator it represents. Its parent atom is of type `kExpressionContainerAtomType`. Its child atoms are of type `kOperandAtomType` and describe the operands which the operator is applied to. The operator is applied to each of its child `kOperandAtomType` atoms in order by their atom index. Unary operators must have one `kOperandAtomType` child atom. Binary operators must have two or more `kOperandAtomType` child atoms.

`kOperandAtomType`

This atom is used to describe an operand, such as a constant number, a sprite property, or a variable. Its parent atom type may be of type `kExpressionContainerAtomType`, in which case the expression consists of only a single operand, or of type `kOperatorAtomType` in which case its index is used to define its order within the operators operand list. The

`kOperandAtomType` atom contains a single child atom which can be any of the operand atom types.

Operator Type Constants

You set the ID of a `kOperatorAtomType` atom to one of the following constants to associate it with a particular operator type. Boolean operations are applied to two operands at a time. If the operand list contains more than two operands, the operation is performed upon the first two and the result is saved. The operation is then repeatedly applied to the current result and the next operand, continuing until no more operands are left. Unary operations are applied to only one operand.

Operands are converted to single-precision, floating-point numbers before the operation is performed.

For the Boolean operations, non-zero operands are considered to be `true`, and operands equal to 0 are considered to be `false`.

The actual numeric result of the Boolean and conditional operations is 1 for `true` and 0 for `false`.

Constant descriptions

<code>kOperatorAdd</code>	This binary operator adds its two operands together producing a numeric result.
<code>kOperatorSubtract</code>	This binary operator subtracts its second operand from its first producing a numeric result.
<code>kOperatorMultiply</code>	This binary operator multiplies its two operands together producing a numeric result.
<code>kOperatorDivide</code>	This binary operator divides its first operand by its second producing a numeric result. If the second operand is 0, a result of 0 is returned.
<code>kOperatorOr</code>	This Boolean binary operator performs a Boolean OR operation. If either of the two operands are true, then the result is <code>true</code> , otherwise the result is <code>false</code> .
<code>kOperatorAnd</code>	This Boolean binary operator performs a Boolean AND operation. If both of the two operands are true, then the result is <code>true</code> , otherwise the result is <code>false</code> .

<code>kOperatorNot</code>	This Boolean unary operator performs a Boolean NOT operation on its single operand. If its operand is <code>true</code> , the result is <code>false</code> , otherwise the result is <code>true</code> .
<code>kOperatorLessThan</code>	This comparison binary operator compares its first operand to its second. If the first is less than the second, the result is <code>true</code> , otherwise the result is <code>false</code> .
<code>kOperatorLessThanEqualTo</code>	This comparison binary operator compares its first operand to its second. If the first is less than or equal to the second, the result is <code>true</code> , otherwise the result is <code>false</code> .
<code>kOperatorEqualTo</code>	This comparison binary operator compares its first operand to its second. If the first is equal to the second, the result is <code>true</code> , otherwise the result is <code>false</code> .
<code>kOperatorNotEqualTo</code>	This comparison binary operator compares its first operand to its second. If the first is not equal to the second, the result is <code>true</code> , otherwise the result is <code>false</code> .
<code>kOperatorGreaterThan</code>	This comparison binary operator compares its first operand to its second. If the first is greater than the second, the result is <code>true</code> , otherwise the result is <code>false</code> .
<code>kOperatorGreaterThanEqualTo</code>	This comparison binary operator compares its first operand to its second. If the first is greater than or equal to the second, the result is <code>true</code> , otherwise the result is <code>false</code> .
<code>kOperatorModulo</code>	This binary operator divides its first operand by its second. The remainder is returned as a numeric result. If the second operand is 0, a result of 0 is returned.
<code>kOperatorIntegerDivide</code>	This binary operator divides its first operand by its second. The remainder is truncated and the root is returned as a numeric result. If the second operand is 0, a result of 0 is returned.
<code>kOperatorAbsoluteValue</code>	This unary operator returns its single operand as a positive numeric number. Positive operands are simply returned; negative operands are multiplied by negative 1 to be made positive, and returned.

`kOperatorNegate` This unary operator multiplies its single operand by negative 1, producing a numeric result.

Operand Type Constants

Operands are used as part of an expression. Operands all return a floating-point number. Most operands return a specific property of their target.

Add one of these atoms as a child atom of a `kOperandAtomType` atom to specify the type of operand. Targets and parameters are specified in the same way that they are for actions. Most of the operands do not need any parameters.

Operands for a Movie

Operands for a movie return the current value of a particular property of a movie.

`kOperandMovieVolume`

No Params

The target movie's current volume level is returned.

`kOperandMovieRate`

No Params

The target movie's current rate is returned.

`kOperandMovieIsLooping`

No Params

If the target movie is in looping mode a value of 1 is returned; otherwise, a value of 0 is returned.

`kOperandMovieLoopIsPalindrome`

No Params

If the target movie is in palindrome mode, a looping value of 1 is returned; otherwise, a value of 0 is returned.

`kOperandMovieTime`

No Params

The target movie's current time value is returned.

Operands for Any Tracks

`kOperandTrackEnabled`

No Params

The target track's enabled state is returned. A value of 1 is returned if it is enabled, and 0 if it is not.

Operands Targeting Spatial Tracks

Operands targeting spatial tracks return the current value of a particular property of a movie. You can use, for example, `kOperandTrackWidth` and `kOperandTrackHeight` to center a sprite within a track.

`kOperandTrackLayer`

No Params

The target track's layer is returned.

`kOperandTrackWidth`

No Params

The target track's width is returned.

`kOperandTrackHeight`

No Params

The target track's height is returned.

`kOperandMouseLocalHLoc`

No Params

The horizontal location of the mouse in the local coordinate system of the target track is returned.

`kOperandMouseLocalVLoc`

No Params

The vertical location of the mouse in the local coordinate system of the target track is returned.

Operands Targeting Sound Tracks

`kOperandTrackVolume`

No Params

The current volume level of the target track is returned.

`kOperandTrackBalance`

No Params

The current balance setting of the target track is returned.

Operands for Sprites in a Sprite Track

`kOperandSpriteBoundsLeft`

No Params

The left side of the target sprite's bounding box in its track's local coordinate system is returned. See "Sprite's Bounding Box" (page 24) for a description of a sprite's bounding box.

`kOperandSpriteBoundsTop`

No Params

The top of the target sprite's bounding box in its track's local coordinate system is returned. See "Sprite's Bounding Box" (page 24) for a description of a sprite's bounding box.

`kOperandSpriteBoundsRight`

No Params

The right side of the target sprite's bounding box in its track's local coordinate system is returned. See "Sprite's Bounding Box" (page 24) for a description of a sprite's bounding box.

`kOperandSpriteBoundsBottom`

No Params

The bottom of the target sprite's bounding box in its track's local coordinate system is returned. See "Sprite's Bounding Box" (page 24) for a description of a sprite's bounding box.

`kOperandSpriteImageIndex`

No Params

The current image index of the target sprite is returned.

`kOperandSpriteVisible`

No Params

The current value of the target sprite's visible property is returned. The value is 1 if it is visible, and 0 if it is invisible.

`kOperandSpriteLayer`

No Params

The layer of the target sprite is returned.

`kOperandSpriteID`**No Params**

The ID of the target sprite is returned. Note that if the ID has a value not expressible in a floating-point number, the results will not be the same.

`kOperandSpriteIndex`**No Params**

The index of the target sprite is returned.

`kOperandSpriteFirstCornerX`**No Params**

The x coordinate of the target sprite's first corner is returned. See "Sprites' Four Corners" (page 25) for a description of a sprite's four corners.

`kOperandSpriteFirstCornerY`**No Params**

The y coordinate of the target sprite's first corner is returned. See "Sprites' Four Corners" (page 25) for a description of a sprite's four corners.

`kOperandSpriteSecondCornerX`**No Params**

The x coordinate of the target sprite's second corner is returned. See "Sprites' Four Corners" (page 25) for a description of a sprite's four corners.

`kOperandSpriteSecondCornerY`**No Params**

The y coordinate of the target sprite's second corner is returned. See "Sprites' Four Corners" (page 25) for a description of a sprite's four corners.

`kOperandSpriteThirdCornerX`**No Params**

The x coordinate of the target sprite's third corner is returned. See "Sprites' Four Corners" (page 25) for a description of a sprite's four corners.

`kOperandSpriteThirdCornerY`**No Params**

The y coordinate of the target sprite's third corner is returned. See "Sprites' Four Corners" (page 25) for a description of a sprite's four corners.

`kOperandSpriteFourthCornerX`

No Params

The x coordinate of the target sprite's fourth corner is returned. See "Sprites' Four Corners" (page 25) for a description of a sprite's four corners.

`kOperandSpriteFourthCornerY`

No Params

The y coordinate of the target sprite's fourth corner is returned. See "Sprites' Four Corners" (page 25) for a description of a sprite's four corners.

`kOperandSpriteImageRegistrationPointX`

No Params

The x coordinate of the target sprite's current image's registration point is returned.

`kOperandSpriteImageRegistrationPointY`

No Params

The y coordinate of the target sprite's current image's registration point is returned.

Operands for a Sprite Track

`kOperandSpriteTrackNumSprites`

No Params

The current number of sprites in the target sprite track is returned. Since the number of sprites is defined by a sprite media key frame sample, this depends on the current movie time.

`kOperandSpriteTrackNumImages`

No Params

The current number of images for the target sprite track is returned. Since the number of sprite images is defined by a sprite media key frame sample, this depends on the current movie time.

`kOperandSpriteTrackVariable`

`Param1: [QTAtomID variableID]`

Returns the value of the variable with the specified ID for the target sprite track. If the variable has never been set with a set variable action, the value returned is 0.

Operands for a QuickTime VR Track

`kOperandQTVRPanAngle`

No Params

Returns the current pan angle for the target QuickTime VR track.

`kOperandQTVRTiltAngle`

No Params

Returns the current tilt angle for the target QuickTime VR track.

`kOperandQTVRFieldOfView`

No Params

Returns the current field of view for the target QuickTime VR track.

`kOperandQTVRNodeID`

No Params

Returns the current node ID for the target QuickTime VR track. Note that if the ID has a value not expressable in a floating-point number, the results will not be the same.

Operands That Have No Target

`kOperandExpression`

`[(kExpressionAtoms)]`

This operand allows for nesting of sub expressions within expressions. Add a child atom of type `kExpressionContainerAtomType` which contains the expression atoms. This expression is evaluated and is returned as the result of the operand.

`kOperandConstant`

`[float theConstant]`

The constant is returned as the result of the operand. Set the leaf data of this atom to the desired constant using a single precision floating point number.

`kOperandKeyIsDown`

Param1: [UInt16 modifierKeyFlags]
Param2: [UInt8 lowerCaseAsciiCharCode]

This operand returns 1 if the key with the specified lower case ASCII character code is currently pressed, and 0 if it is not pressed. If you specify modifier keys, these keys must also be pressed for 1 to be returned. Note that if extra modifier keys are pressed, a positive indication will still be returned. The logic is “are these keys currently pressed?”, not “are only these keys pressed?”.

Additionally, you may determine if the mouse button is pressed.

You may pass 0 for modifier keys if you don’t care about the modifier keys, and 0 for the ASCII character if you only care about the modifier keys.

The following flags may be used from `Events.h`. Note the mapping of the flags for Windows machines. Also, be warned that using the Control key may be a bad idea if you plan on using the movie on a Windows machine due to the fact that it is mapped to the Alt key, which is used by the Windows operating system.

<code>btnState</code>	The mouse button is pressed. Windows: the left mouse button is pressed.
<code>cmdKey</code>	The command key is pressed. Windows: the Ctrl key is pressed.
<code>shiftKey</code>	The Shift key is pressed.
<code>alphaLock</code>	The Caps Lock key is pressed.
<code>optionKey</code>	The option key is pressed. Under Windows: both the Ctrl and the Alt keys are pressed.
<code>controlKey</code>	The control key is pressed. Windows: the Alt key is pressed.

`kOperandRandom`

Param1: `minimumValue`
Param2: `maximumValue`

A random number that is greater than or equal to the minimum value, and less than or equal to the maximum value is generated and returned as the result of this operand. You may use negative or positive numbers.

Additions to the Standard Movie Controller

`mcActionExecuteAllActionsForQTEvent`

This allows your application to filter QuickTime events for which the movie has actions. This is sent before any actions are executed. Return `true` from your filter proc if you do not want the actions to be executed; return `false` if you do want the actions executed.

The parameter is a `ResolvedQTEventSpecPtr`. The type of QuickTime event may be determined from the `QTAtomSpec` portion. The atom is either a `kQTEventFrameLoaded` atom, or a `kQTEventType` atom in which case its ID specifies the type of QuickTime event.

`mcActionExecuteOneActionForQTEvent`

This allows your application to filter individual actions which are about to be executed in response to a QuickTime event. This is sent before the action is executed. Return `true` from your filter proc if you do not want the action to be executed, return `false` if you do want the action executed.

The parameter is a `ResolvedQTEventSpecPtr`. The type of action may be determined from the `QTAtomSpec` portion. The atom is a `kAction` atom. The type of action is specified by the leaf data of its child atom of type `kWhichAction`.

`mcActionShowMessageString`

This allows your application to receive and display a message from the movie. Currently, the only way this message is sent is by executing a `kActionDebugStr` action.

The parameter is a `StringPtr` which contains the message. Although named `DebugStr`, the action can be used for any type of string-based message.

Data Types

The `QTEventRecord` data type is used by the `kActionSendQTEventToSprite` action.

```
struct QTEventRecord {
    long                version;
    OSType              eventType;
    Point               where;
    long                flags;
};
typedef struct QTEventRecord QTEventRecord;

typedef QTEventRecord *    QTEventRecordPtr;
```

CHAPTER 6

Wired Movies API Reference

Sprite Toolbox API Reference

This chapter describes the constants and functions available to your application in the sprite toolbox. It also describes the functions provided by the Movie Toolbox for sprite support.

For more information about using the sprite toolbox, refer to Chapter 4, “Using the Sprite Toolbox to Create Sprite Animations.”

Constants

Background Sprites

You assign the following constant to a sprite’s `kSpritePropertyLayer` property to designate the sprite as a background sprite.

```
enum {
    kBackgroundSpriteLayerNum    = 32767
};
```

Flags for Sprite Hit Testing

You can pass the following flags to `SpriteWorldHitTest` (page 153) and `SpriteHitTest` (page 157) to control sprite hit testing.

```
enum {
    spriteHitTestBounds          = 1L << 0,
    spriteHitTestImage           = 1L << 1,
    spriteHitTestInvisibleSprites = 1L << 2,
```

Sprite Toolbox API Reference

```

    spriteHitTestIsClick                = 1L << 3,
    spriteHitTestLocInDisplayCoordinates = 1L << 4
};

```

Flag descriptions

`spriteHitTestBounds`

The specified location must be within the sprite's bounding box.

`spriteHitTestImage`

If both this flag and `spriteHitTestBounds` are set, the specified location must be within the shape of the sprite's image.

`spriteHitTestInvisibleSprites`

This flag enables invisible sprites to be hit tested.

`spriteHitTestIsClick`

This flag is for codecs that want mouse events, such as the ripple codec.

`spriteHitTestLocInDisplayCoordinates`

You set this flag if you want to pass a display coordinate point to `SpriteHitTest`, such as returned by the Mac OS Toolbox routine `getMouse`.

Sprite Properties

The following constants represent the different properties of a sprite. When you call `SetSpriteProperty` (page 159) to set a sprite property, you pass one of these constants to specify the property you wish to modify.

```

enum {
    kSpritePropertyMatrix                = 1,
    kSpritePropertyImageDescription      = 2,
    kSpritePropertyImageDataPtr          = 3,
    kSpritePropertyVisible                = 4,
    kSpritePropertyLayer                  = 5,
    kSpritePropertyGraphicsMode          = 6,
    kSpritePropertyImageIndex            = 100
};

```


Constant descriptions`kSpritePropertyMatrix`

A matrix of type `MatrixRecord` that defines the sprite's display coordinate system.

`kSpritePropertyImageDescription`

An image description handle that describes the sprite's image data. This must be valid as long as the sprite uses it. The caller owns the storage. The sprite toolbox does not copy this data.

`kSpritePropertyImageDataPtr`

A pointer to the sprite's image data. This must be valid as long as the sprite uses it. The caller owns the storage. The sprite toolbox does not copy this data.

`kSpritePropertyVisible`

A Boolean value that indicates whether the sprite is visible.

`kSpritePropertyLayer`

A short integer value that defines the sprite's layer. You set this property to `kBackgroundSpriteLayerNum` to designate the sprite as a background sprite.

`kSpritePropertyGraphicsMode`

A `ModifierTrackGraphicsModeRecord` value that specifies the graphics mode to be used when drawing the sprite.

`kSpritePropertyImageIndex`

In a sprite track, the index of the sprite's image in the pool of shared images.

Flags for `SpriteWorldIdle`

You can pass the following flags as input to `SpriteWorldIdle` (page 151) to control drawing of the sprite world.

```
enum {
    kOnlyDrawToSpriteWorld = 1L << 0,
    kSpriteWorldPreFlight   = 1L << 1
};
```

Flag descriptions`kOnlyDrawToSpriteWorld`

You set this flag to indicate that drawing should take place in the sprite world only; drawing to the final destination should be suppressed.

`kSpriteWorldPreFlight`

You can set this flag to determine whether the sprite world has any invalid areas that need to be drawn. If so, the `SpriteWorldIdle` function returns the `kSpriteWorldNeedsToDraw` flag in the `flagsOut` parameter.

The following flags may be returned in the `flagsOut` parameter of `SpriteWorldIdle` (page 151).

```
enum {
    kSpriteWorldDidDraw          = 1L << 0,
    kSpriteWorldNeedsToDraw     = 1L << 1
};
```

Flag descriptions`kSpriteWorldDidDraw`

If set, this flag indicates that `SpriteWorldIdle` updated the sprite world.

`kSpriteWorldNeedsToDraw`

If set, this flag indicates that the sprite world has invalid areas that need to be drawn.

Sprite and Sprite World Identifiers

The sprite world and sprite data structures are private data structures. You identify a sprite world or a sprite data structure to the sprite toolbox by means of a data type that is supplied by the sprite toolbox. The following data types are currently defined:

`Sprite`

Specifies the sprite for an operation. Your application obtains a sprite identifier when you create a new sprite by calling `NewSprite` (page 154).

`SpriteWorld`

Specifies the sprite world for an operation. Your application obtains a sprite world identifier when you create a sprite

world by calling `NewSpriteWorld` (page 147).

Sprite Toolbox Functions

This section describes the functions provided by the Movie Toolbox for sprite support.

Sprite World Functions

This section describes functions that you use to create and manipulate sprite worlds.

NewSpriteWorld

The `NewSpriteWorld` function creates a new sprite world.

```
pascal OSErr NewSpriteWorld (SpriteWorld *newSpriteWorld,
                             GWorldPtr destination,
                             GWorldPtr spriteLayer,
                             RGBColor *backgroundColor,
                             GWorldPtr background);
```

`newSpriteWorld`

Contains a pointer to a field that is to receive the new sprite world's identifier. On return, this field contains the identifier for the newly created sprite world.

`destination`

Contains a pointer to a graphics world to be used as the destination.

`spriteLayer`

Contains a pointer to a graphics world to be used as the sprite layer.

`backgroundColor`

Contains a pointer to an RGB color to be used as the background color. If you pass a background graphics world to this function by setting the `background` parameter, you can set this parameter to `nil`.

`background`

Contains a pointer to a graphics world to be used as the background. If you pass a background color to this function by setting the `backgroundColor` parameter, you can set this parameter to `nil`.

DISCUSSION

You call this function to create a new sprite world with associated destination and sprite layer graphics worlds, and either a background color or a background graphics world. Once created, you can manipulate the sprite world and add sprites to it using other sprite toolbox functions. The sprite world created by this function has an identity matrix. The sprite world does not have a clip shape.

The `newSpriteWorld`, `destination`, and `spriteLayer` parameters are all required. You should specify a background color, a background graphics world, or both. You should not pass `nil` for both parameters. If you specify both a background graphics world and a background color, the sprite world is filled with the background color before the background sprites are drawn. If no background color is specified, black is the default. If you specify a background graphics world, it should have the same dimensions and depth as the graphics world specified by `spriteLayer`. If you draw to the graphics worlds associated with a sprite world using standard `QuickDraw` and `QuickTime` functions, your drawing is erased by the sprite world's background color.

Before calling `NewSpriteWorld`, you should call `LockPixels` on the pixel maps of the sprite layer and background graphics worlds. These graphics worlds must remain valid and locked for the lifetime of the sprite world. The sprite world does not own the graphics worlds that are associated with it; it is the caller's responsibility to dispose of the graphics worlds when they are no longer needed.

RESULT CODES

<code>noErr</code>	0	No error
<code>paramErr</code>	-50	Invalid parameter specified
<code>memFullErr</code>	-108	Not enough memory available
<code>gWorldsNotSameDepthAndSizeErr</code>	-2066	Dimensions and pixel depth of the two graphics worlds do not match

DisposeSpriteWorld

The `DisposeSpriteWorld` function disposes of a sprite world.

```
pascal void DisposeSpriteWorld (SpriteWorld theSpriteWorld);
```

```
theSpriteWorld
```

Specifies the sprite world to dispose.

DISCUSSION

You call this function to dispose of a sprite world created by the `NewSpriteWorld` function. This function also disposes of all of the sprites associated with the sprite world. This function does not dispose of the graphics worlds associated with the sprite world. It is safe to pass `nil` to this function.

SetSpriteWorldClip

The `SetSpriteWorldClip` function sets a sprite world's clip shape to the specified region.

```
pascal OSErr SetSpriteWorldClip (SpriteWorld theSpriteWorld,
                                RgnHandle clipRgn);
```

```
theSpriteWorld
```

Specifies the sprite world for this operation.

```
clipRgn
```

Specifies the new clip shape for the sprite world.

DISCUSSION

You call this function to change the clip shape of a sprite world. You may pass a value of `nil` for the `clipRgn` parameter to indicate that there is no longer a clip shape for the sprite world. This means that the whole area is drawn.

The clip shape should be specified in the sprite world's source space, the coordinate system of the sprite layer's graphics world before the sprite world's matrix is applied to it. The specified region is owned by the caller and is not copied by this function.

RESULT CODES

<code>noErr</code>	0	No error
<code>paramErr</code>	-50	Invalid parameter specified

SetSpriteWorldMatrix

The `SetSpriteWorldMatrix` function sets a sprite world's matrix to the specified matrix.

```
pascal OSErr SetSpriteWorldMatrix (SpriteWorld theSpriteWorld,
                                   const MatrixRecord *matrix);
```

`theSpriteWorld`

Specifies the sprite world for this operation.

`matrix`

Contains a pointer to the new matrix for the sprite world.

DISCUSSION

You call this function to change the matrix of a sprite world. You may pass a value of `nil` for the `matrix` parameter to set the sprite world's matrix to an identity matrix.

Transformations, including translation, scaling, rotation, skewing, and perspective, are all supported in QuickTime 3.

RESULT CODES

<code>noErr</code>	0	No error
<code>paramErr</code>	-50	Invalid parameter specified

SpriteWorldIdle

The `SpriteWorldIdle` function allows a sprite world to update its invalid areas.

```
pascal OSErr SpriteWorldIdle (SpriteWorld theSpriteWorld,
                             long flagsIn,
                             long *flagsOut);
```

`theSpriteWorld`

Specifies the sprite world for this operation.

`flagsIn`

Contains flags describing actions that may take place during the idle.

`flagsOut`

On return, contains a pointer to flags describing actions that took place during the idle.

DISCUSSION

You call this function to allow a sprite world the opportunity to redraw its invalid areas. This is the only function that causes drawing to occur; you should call it as often as is necessary.

The `flagsIn` parameter contains flags that describe allowable actions during the idle period. For the default behavior, you should set the value of this parameter to 0. The `flagsOut` parameter is optional; if you do not need the information returned by this parameter, set the value of this parameter to `nil`.

Typically, you would make changes in perspective for a number of sprites and then call `SpriteWorldIdle` to redraw the changed sprites.

RESULT CODES

<code>noErr</code>	0	No error
<code>paramErr</code>	-50	Invalid parameter specified

InvalidateSpriteWorld

The `InvalidateSpriteWorld` function invalidates a rectangular area of a sprite world.

```
pascal OSErr InvalidateSpriteWorld (SpriteWorld theSpriteWorld,
                                     Rect *invalidArea);
```

`theSpriteWorld`

Specifies the sprite world for this operation.

`invalidArea`

Contains a pointer to the rectangular area that should be invalidated.

DISCUSSION

Typically, your application calls this function when the sprite world's destination window receives an update event. Invalidating an area of the sprite world will cause the area to be redrawn the next time that `SpriteWorldIdle` is called.

The invalid rectangle pointed to by the `invalidArea` parameter should be specified in the sprite world's source space, the coordinate system of the sprite layer's graphics world before the sprite world's matrix is applied to it. To invalidate the entire sprite world, pass `nil` for this parameter.

When you modify sprite properties, invalidation takes place automatically; you do not need to call `InvalidateSpriteWorld`.

RESULT CODES

<code>noErr</code>	0	No error
<code>paramErr</code>	-50	Invalid parameter specified

SpriteWorldHitTest

The `SpriteWorldHitTest` function determines whether any sprites are at a specified location in a sprite world.

```
pascal OSErr SpriteWorldHitTest (SpriteWorld theSpriteWorld,
                                long flags,
                                Point loc,
                                Sprite *spriteHit);
```

`theSpriteWorld`

Specifies the sprite world for this operation.

`flags`

Specifies flags to control the hit testing operation. These flags are described in “Flags for Sprite Hit Testing” (page 143).

`loc`

Specifies a point in the sprite world’s display space to test for the existence of a sprite.

`spriteHit`

Contains a pointer to a field that is to receive a sprite identifier. On return, this field contains the identifier of the frontmost sprite at the location specified by `loc`. If no sprite exists at the location, the function sets the value of this parameter to `nil`.

DISCUSSION

You call this function to determine whether any sprites exist at a specified location in a sprite world’s display coordinate system. If you are drawing the sprite world in a window, you should call `GlobalToLocal` to convert the location to your window’s local coordinate system before passing it to `SpriteWorldHitTest`.

You use the `spriteHitTestBounds` and `spriteHitTestImage` flags in the `flags` parameter to control the hit test operation. Set the `spriteHitTestBounds` flag to check if there has been a hit anywhere within the sprite’s bounding box. Set the `spriteHitTestImage` flag to check if there has been a hit anywhere within the sprite image.

A hit testing operation does not occur unless you pass one of the flags, either `SpriteHitTestBound` or `SpriteHitTestImage`. You can add other flags as needed.

RESULT CODES

noErr	0	No error
paramErr	-50	Invalid parameter specified

DisposeAllSprites

The `DisposeAllSprites` function disposes all sprites associated with a sprite world.

```
pascal void DisposeAllSprites (SpriteWorld theSpriteWorld);
```

```
theSpriteWorld
```

Specifies the sprite world for this operation.

DISCUSSION

This function calls the `DisposeSprite` function for each sprite associated with the sprite world.

Sprite Functions

This section describes functions that you use to create and manipulate sprites.

NewSprite

The `NewSprite` function creates a new sprite in the specified sprite world.

```
pascal OSErr NewSprite (Sprite *newSprite,
                        SpriteWorld itsSpriteWorld,
                        ImageDescriptionHandle idh,
                        Ptr imageDataPtr,
                        MatrixRecord *matrix,
                        Boolean visible,
                        short layer);
```

Sprite Toolbox API Reference

<code>newSprite</code>	Contains a pointer to field that is to receive the new sprite's identifier. On return, this field contains the identifier of the newly created sprite.
<code>itsSpriteWorld</code>	Specifies the sprite world with which the new sprite should be associated.
<code>idh</code>	Contains a handle to an image description of the sprite's image.
<code>imageDataPtr</code>	Contains a pointer to the sprite's image data.
<code>matrix</code>	Contains a pointer to the sprite's matrix. If you pass <code>nil</code> for the <code>matrix</code> parameter, an identity matrix is assigned to the sprite.
<code>visible</code>	Specifies whether the sprite is visible.
<code>layer</code>	Specifies the sprite's layer.

DISCUSSION

You call this function to create a new sprite associated with a sprite world. Once you have created the sprite, you can manipulate it using `SetSpriteProperty` (page 159).

The `newSprite`, `itsSpriteWorld`, `visible`, and `layer` parameters are required. Sprites with lower layer values appear in front of sprites with higher layer values. If you want to create a sprite that is drawn to the background graphics world, you should specify the constant `kBackgroundSpriteLayerNum` for the `layer` parameter.

You can defer assigning image data to the sprite by passing `nil` for both the `idh` and `imageDataPtr` parameters. If you choose to defer assigning image data, you must call `SetSpriteProperty` to assign the image description handle and image data to the sprite before the next call to `SpriteWorldIdle`. The caller owns the image description handle and the image data pointer; it is the caller's responsibility to dispose of them after it disposes of a sprite.

RESULT CODES

<code>noErr</code>	0	No error
<code>paramErr</code>	-50	Invalid parameter specified
<code>memFullErr</code>	-108	Not enough memory available

DisposeSprite

The `DisposeSprite` function disposes of a sprite.

```
pascal void DisposeSprite (Sprite theSprite);
```

`theSprite` The sprite for this operation.

DISCUSSION

You call this function to dispose of a sprite created by the `NewSprite` function. The image description handle and image data pointer associated with the sprite are not disposed by this function.

InvalidateSprite

The `InvalidateSprite` function invalidates the portion of a sprite's sprite world that is occupied by the sprite.

```
pascal void InvalidateSprite (Sprite theSprite);
```

`theSprite` The sprite for this operation.

DISCUSSION

In most cases, you do not need to call this function. When you call the `SetSpriteProperty` function to modify a sprite's properties, `SetSpriteProperty` takes care of invalidating the appropriate regions of the sprite world. However, you might call this function if you change a sprite's image data, but retain the same image data pointer.

SpriteHitTest

The `SpriteHitTest` function determines whether a location in a sprite's display coordinate system intersects the sprite.

```
pascal OSErr SpriteHitTest (Sprite theSprite,
                           long flags,
                           Point loc,
                           Boolean *wasHit);
```

<code>theSprite</code>	Specifies the sprite for this operation.
<code>flags</code>	Specifies flags to control the hit testing operation. These flags are described in “Flags for Sprite Hit Testing” (page 143).
<code>loc</code>	Specifies a point in the sprite world's display space to test for the existence of a sprite.
<code>wasHit</code>	Contains a pointer to a Boolean. On return, the value of the Boolean is <code>true</code> if the sprite is at the specified location.

DISCUSSION

You call this function to determine whether a sprite exists at a specified location in the sprite's display coordinate system. This function is useful for hit testing a subset of the sprites in a sprite world and for detecting multiple hits for a single location.

You should apply the sprite world's matrix to the location before passing it to `SpriteHitTest`. To convert a location to local coordinates, you should use the `GlobalToLocal` function to convert the location to your window's local coordinate system and then apply the inverse of the sprite world's matrix to the location.

You use the `spriteHitTestBounds` and `spriteHitTestImage` flags in the `flags` parameter to control the hit test operation. Set the `spriteHitTestBounds` flag to check if there has been a hit anywhere within the sprite's bounding box. Set the `spriteHitTestImage` flag to check if there has been a hit anywhere within the sprite image.

RESULT CODES

noErr	0	No error
paramErr	-50	Invalid parameter specified

GetSpriteProperty

The `GetSpriteProperty` function retrieves the value of the specified sprite property.

```
pascal OSErr GetSpriteProperty (Sprite theSprite,
                                long propertyType,
                                void *propertyValue);
```

theSprite	Specifies the sprite for this operation.
propertyType	Specifies the property whose value should be retrieved.
propertyValue	A pointer to a variable that will hold the value on return.

DISCUSSION

You call this function to retrieve a value of a sprite property. You set the `propertyType` parameter to the property you want to retrieve. The following table lists the sprite properties and the data types of the corresponding property values.

Sprite Property	Data Type
kSpritePropertyMatrix	MatrixRecord
kSpritePropertyImageDescription	ImageDescriptionHandle
kSpritePropertyImageDataPtr	Ptr
kSpritePropertyVisible	Boolean
kSpritePropertyLayer	short
kSpritePropertyGraphicsMode	ModifierTrackGraphicsModeRecord

In the case of the `kSpritePropertyImageDescription` and `kSpritePropertyImageDataPtr` properties, this function does not return a copy of the data; rather, the pointers returned are references to the sprite's data.

RESULT CODES

noErr	0	No error
invalidSpritePropertyErr	-2065	The specified sprite property does not exist

SetSpriteProperty

The `SetSpriteProperty` function sets the specified property of a sprite.

```
pascal OSErr SetSpriteProperty (Sprite theSprite,
                                long propertyType,
                                void *propertyValue);
```

<code>theSprite</code>	Specifies the sprite for this operation.
<code>propertyType</code>	Specifies the property to be set.
<code>propertyValue</code>	Specifies the new value of the property.

DISCUSSION

You animate a sprite by modifying its properties. You call this function to modify a property of a sprite. This function invalidates the sprite's sprite world as needed.

You set the `propertyType` parameter to the property you want to modify. Depending on the property type, you set the `propertyValue` parameter to either a pointer to the property value or the property value itself, cast as a `void*`.

The following table lists the sprite properties and the data types of the corresponding property values.

Sprite Property	Data Type
<code>kSpritePropertyMatrix</code>	<code>MatrixRecord *</code>
<code>kSpritePropertyImageDescription</code>	<code>ImageDescriptionHandle</code>
<code>kSpritePropertyImageDataPtr</code>	<code>Ptr</code>

Sprite Toolbox API Reference

Sprite Property

kSpritePropertyVisible
kSpritePropertyLayer
kSpritePropertyGraphicsMode

Data Type

Boolean
short
ModifierTrackGraphicsModeRecord *

RESULT CODES

noErr	0	No error
memFullErr	-108	Not enough memory available
invalidSpritePropertyErr	-2065	Specified sprite property does not exist

Sprite Media Handler API Reference

This chapter provides an API reference of constants and functions available to your application. The chapter also describes the new atom types added to QuickTime 4.

For more conceptual information about sprite media handlers, refer to Chapter 2, “The Sprite Media Handler.”

Constants

Action Media Format Atoms

The sprite track’s sample format enables you to store the atoms necessary to describe action lists which are executed in response to QuickTime events. Appendix B defines a grammar for constructing valid action sprite samples, which may include complex expressions.

Both key frame samples and override samples support the sprite action atoms. Override samples override actions at the QuickTime event level. In effect, what you do by overriding is to completely replace one event handler and all its actions with another. The sprite track’s `kSpriteTrackPropertySampleFormat` property has no effect on how actions are performed. The behavior is similar to the default `kKeyFrameAndSingleOverride` format where, if in a given override sample there is no handler for the event, the key frame’s handler is used, if there is one.

Action Sprites Media Format Extensions

This section describes all of the atom types and IDs which are used to extend the sprite track’s media format, enabling the new action sprite capabilities.

Chapter 3, “Authoring Wired Movies and Sprite Animations,” describes how to create basic sprite samples with images, sprites, and their properties. A complete description of the grammar for sprite media handler samples, including action sprite extensions, is included in Appendix B.

IMPORTANT

There are also new sprite track property atoms. In particular, you must set the `kSpriteTrackPropertyHasActions` track property in order for your sprite actions to be executed. ▲

Sprite Track Property Atoms

The following section describes sprite track property atoms. Track property atoms are applied to the whole track, not just to a single sample. See Chapter 3, “Authoring Wired Movies and Sprite Animations,” for information on other track property atoms and how to add them to the sprite track’s media.

Constant descriptions

`kSpriteTrackPropertyHasActions`

You must add an atom of this type with its leaf data set to `true` if you want the movie controller to execute the actions in your sprite track’s media. The atom’s leaf data is of type `Boolean`. The default value is `false`, so it is very important to add an atom of this type if you want interactivity to take place.

`kSpriteTrackPropertyQTIdleEventsFrequency`

You must add an atom of this type if you want the Sprites in your sprite track to receive `kQTEventIdle` QuickTime events. The atom’s leaf data is of type `UInt32`. The value is the minimum number of ticks that must pass before the next `QTIdle` event is sent. Each tick is 1/60th of one second. To specify “Idle as fast as possible,” set the value to 0. The default value is `kNoQTIdleEvents` which means don’t send any idle events.

It is possible that for small idle event frequencies, the movie will not be able to keep up, in which case they will be sent as fast as possible.

Since sending idle events takes up some time, it is best to specify the largest frequency that produces the results that you desire, or `kNoQTIdleEvents` if you do not need them.

`kSpriteTrackPropertyVisible`

You may cause the entire sprite track to be invisible by setting the value of this `Boolean` property to `false`. This is useful for using a sprite track as a hidden button track—for example, placing an invisible sprite track over a `Video Track` would allow the characters in the video to be clicked on. The default value is visible (`true`).

`kSpriteTrackPropertyScaleSpritesToScaleWorld`

You may cause each sprite to be rescaled when the sprite track is resized by setting the value of this `Boolean` property to `true`. Setting this property can improve the drawing performance and quality of a scaled sprite track. This is particularly useful for sprite images compressed with codecs which are resolution-independent, such as the `Curve` codec. The default value for this property is `false`.

Atom Types

`kSpriteUsesImageIDsAtomType`

This atom allows a sprite to specify which images it uses—in other words, the subset of images that its `imageIndex` property will ever refer to.

You add an atom of type `kSpriteUsesImageIDsAtomType` as a child of a `kSpriteAtomType` atom, setting its leaf data to an array of `QTAtomIDs`. This array contains the IDs of the images used, not the indices.

Note

Although `QuickTime` does not currently use this atom internally, tools that edit sprite media can use the information provided to optimize certain operations, such as `Cut`, `Copy`, and `Paste`. ♦

`kSpriteImageRegistrationAtomType`

Sprite images have a default registration point of 0, 0. To specify a different point, you add an atom of type

`kSpriteImageRegistrationAtomType` as a child atom of the `kSpriteImageAtomType` and set its leaf data to a `FixedPoint` value with the desired registration point.

`kSpriteImageGroupIDAtomType`

Prior to QuickTime 3, sprites could only display images with the same image description. This restriction has been relaxed in QuickTime 3, but you must assign group IDs to sets of equivalent images in your key frame sample. For example, if the sample contains 10 images where the first 2 images are equivalent, and the last 8 images are equivalent, then you could assign a group ID of 1000 to the first 2 images, and a group ID of 1001 to the last 8 images. This divides the images in the sample into two sets. The actual ID does not matter, it just needs to be a unique positive integer.

Each image in a sprite media key frame sample is assigned to a group. You add an atom of type

`kSpriteImageGroupIDAtomType` as a child of the `kSpriteImageAtomType` atom and set its leaf data to a long containing the group ID.

IMPORTANT

You must assign group IDs to your sprite sample if you want a sprite to display images with non-equivalent image descriptions (i.e., images with different dimensions). ▲

New Atom Types in QuickTime 4

The following new atom types have been added to QuickTime 4. You use these atom types to specify that an image is referenced, and how to access it.

Constant descriptions

`kSpriteImageDataRefAtomType`

You add this atom as a child of the `kSpriteImageAtomType` atom instead of a `kSpriteImageDataAtomType`. Its ID should be 1. Its data should contain the data reference (similar to the `dataRef` parameter of `GetDataHandler()`).

`kSpriteImageDataRefTypeAtomType`

You add this atom as a child of the `kSpriteImageAtomType` atom. Its ID should be 1. Its data should contain the data reference type (similar to the `dataRefType` parameter of `GetDataHandler()`).

`kSpriteImageDefaultImageIndexAtomType`

You may optionally add this atom as a child of the `kSpriteImageAtomType` atom. Its ID should be 1. Its data should contain a `short`, which specifies an image index of a traditional image to use while waiting for the referenced image to load.

Sprite Track Formats

The following constants represent formats of a sprite track. The value of the constant indicates how override samples in a sprite track should be interpreted. You set a sprite track's format by creating a `kSpriteTrackPropertySampleFormat` atom.

```
enum {
    kKeyFrameAndSingleOverride    = 1L << 1,
    kKeyFrameAndAllOverrides      = 1L << 2
};
```

Constant descriptions

`kKeyFrameAndSingleOverride`

The current state of the sprite track is defined by the most recent key frame sample and the current override sample. This is the default format.

`kKeyFrameAndAllOverrides`

The current state of the sprite track is defined by the most recent key frame sample and all subsequent override samples up to and including the current override sample.

Sprite Media Atom Types

The following constants represent atom types for sprite media.

Sprite Media Handler API Reference

```
enum {
    kSpriteAtomType                = 'sprt',
    kSpriteImagesContainerAtomType = 'imct',
    kSpriteImageAtomType           = 'imag',
    kSpriteImageDataAtomType       = 'imda',
    kSpriteSharedDataAtomType      = 'dflt',
    kSpriteNameAtomType            = 'name'
    kSpriteURLLinkAtomType         = 'url'
    kSpritePropertyMatrix          = 1
    kSpritePropertyVisible         = 4
    kSpritePropertyLayer           = 5
    kSpritePropertyGraphicsMode    = 6
    kSpritePropertyImageIndex      = 101
    kSpritePropertyBackgroundColor = 101
    kSpritePropertyOffscreenBitDepth = 102
    kSpritePropertySampleFormat    = 103
};
```

Constant descriptions

`kSpriteAtomType` **The atom is a parent atom that describes a sprite. It contains atoms that describe properties of the sprite. Optionally, it may also include an atom of type `kSpriteNameAtomType` that defines the name of the sprite.**

`kSpriteImagesContainerAtomType` **The atom is a parent atom that contains atoms of type `kSpriteImageAtomType`.**

`kSpriteImageAtomType` **The atom is a parent atom that contains an atom of type `kSpriteImageDataAtomType`. Optionally, it may also include an atom of type `kSpriteNameAtomType` that defines the name of the image.**

`kSpriteImageDataAtomType` **The atom is a leaf atom that contains image data.**

`kSpriteSharedDataAtomType` **The atom is a parent atom that contains shared sprite data, such as an atom container of type `kSpriteImagesContainerAtomType`.**

`kSpriteNameAtomType` **The atom is a leaf atom that contains the name of a sprite or**

an image. The leaf data is composed of one or more ASCII characters.

`kSpritePropertyImageIndex`

A leaf atom containing the image index property which is of type `short`. This atom is a child atom of the `kSpriteAtom`.

`kSpritePropertyLayer`

A leaf atom containing the layer property which is of type `short`. This atom is a child atom of the `kSpriteAtom`.

`kSpritePropertyMatrix`

A leaf atom containing the matrix property which is of type `MatrixRecord`. This atom is a child atom of the `kSpriteAtom`.

`kSpritePropertyVisible`

A leaf atom containing the visible property which is of type `short`. This atom is a child atom of the `kSpriteAtom`.

`kSpritePropertyGraphicsMode`

A leaf atom containing the matrix property which is of type `ModifierTrackGraphicsModeRecord`. This atom is a child atom of the `kSpriteAtom`.

`kSpritePropertyBackgroundColor`

A leaf atom containing the background color property which is of type `RGBColor`. This atom is used in a sprite track's `MediaPropertyAtom` atom container.

`kSpritePropertyOffscreenBitDepth`

A leaf atom containing the preferred offscreen bitdepth which is of type `short`. This atom is used in a sprite track's `MediaPropertyAtom` atom container.

`kSpritePropertySampleFormat`

A leaf atom containing the sample format property which is of type `short`. This atom is used in a sprite track's `MediaPropertyAtom` atom container.

`kSpriteImageRegistrationAtomType`

Sprite images have a default registration point of 0, 0. To specify a different point, add an atom of type `kSpriteImageRegistrationAtomType` as a child atom of the `kSpriteImageAtomType` and set its leaf data to a `FixedPoint` value with the desired registration point.

`kSpriteImageGroupIDAtomType`

Before QuickTime 3, sprites could only display images with the same image description. This restriction has been relaxed, but you must assign group IDs to sets of equivalent images in your key frame sample. For example, if the sample contains ten images where the first two images are equivalent, and the last eight images are equivalent, then you could assign a group ID of 1000 to the first two images, and a group ID of 1001 to the last eight images. This divides the images in the sample into two sets. The actual ID does not matter, it just needs to be a unique positive integer.

Each image in a sprite media key frame sample is assigned to a group. Add an atom of type

`kSpriteImageGroupIDAtomType` as a child of the `kSpriteImageAtomType` atom and set its leaf data to a long containing the group ID.

The following are new atom types that have been added to QuickTime 4. For each of these, except `kSpriteBehaviorsAtomType`, you fill in this struct [QTSpriteButtonBehaviorStruct] which contains a value for each of the four states.

`kSpriteBehaviorsAtomType, 1`

This is the parent atom of `kSpriteImageBehaviorAtomType`, `kSpriteCursorBehaviorAtomType`, and `kSpriteStatusStringsBehaviorAtomType`. For more information about this new atom type, refer to “Sprite Button Behaviors” (page 46).

`kSpriteImageBehaviorAtomType`

Specifies the `imageIndex`. For more information about this new atom type, refer to “Sprite Button Behaviors” (page 46).

`kSpriteCursorBehaviorAtomType`

Specifies the `cursorID`. For more information about this new atom type, refer to “Sprite Button Behaviors” (page 46).

`kSpriteStatusStringsBehaviorAtomType`

Specifies an ID of a string variable contained in a sprite track to display in the status area of the browser. For more information about this new atom type, refer to “Sprite Button Behaviors” (page 46).

IMPORTANT

You must assign group IDs to your sprite sample if you want a sprite to display images with non-equivalent image descriptions (i.e., images with different dimensions). ▲

Note

All sprite media—specifically the leaf data in the QT atom containers for sample and sprite track properties—should be written in big-endian format. ♦

Sprite Media Handler Functions

This section describes the sprite media handler functions available to your application.

SpriteMediaSetSpriteProperty

The `SpriteMediaSetSpriteProperty` function sets the specified property of a sprite.

```
pascal ComponentResult SpriteMediaSetSpriteProperty (
    MediaHandler mh,
    QTAtomID spriteID,
    long propertyType,
    void* propertyValue);
```

<code>mh</code>	Specifies the sprite media handler for this operation.
<code>spriteID</code>	Specifies the ID of the sprite to be modified.
<code>propertyType</code>	Specifies the property to be set.
<code>propertyValue</code>	Specifies the new value of the property.

DISCUSSION

You call this function to modify a property of a sprite. You set the `propertyType` parameter to the property you want to modify. You set the `spriteID` parameter to the ID of the sprite whose property you want to set.

The type of data you pass for the `propertyValue` parameter depends on the property type. The following table lists the sprite properties and the data types of the corresponding property values.

Sprite Property	Data Type
<code>kSpritePropertyMatrix</code>	<code>MatrixRecord *</code>
<code>kSpritePropertyVisible</code>	<code>short</code>
<code>kSpritePropertyLayer</code>	<code>short</code>
<code>kSpritePropertyGraphicsMode</code>	<code>ModifierTrackGraphicsModeRecord *</code>
<code>kSpritePropertyImageIndex</code>	<code>short</code>

RESULT CODES

<code>noErr</code>	<code>0</code>	No error
<code>invalidSpritePropertyErr</code>	<code>-2065</code>	Specified sprite property does not exist
<code>invalidSpriteIndexErr</code>	<code>-2067</code>	Sprite index is out of range

SpriteMediaGetSpriteProperty

The `SpriteMediaGetSpriteProperty` function retrieves the value of the specified sprite property.

```
pascal ComponentResult SpriteMediaGetSpriteProperty (
    MediaHandler mh,
    QTAtomID spriteID,
    long propertyType,
    void* propertyValue);
```

<code>mh</code>	Specifies the sprite media handler for this operation.
<code>spriteID</code>	Specifies the ID of the sprite for this operation.
<code>propertyType</code>	Specifies the property whose value should be retrieved.

`propertyValue`
On return, contains a pointer to the value of the property.

DISCUSSION

You call this function to retrieve a value of a sprite property. You set the `propertyType` parameter to the property you want to retrieve. You set the `spriteID` parameter to the ID of the sprite whose property you want to retrieve.

On return, the `propertyValue` parameter contains a pointer to the specified property's value; the data type of that value depends on the property. The following table lists the sprite properties and the data types of the corresponding property values.

Sprite Property	Data Type
<code>kSpritePropertyMatrix</code>	<code>MatrixRecord *</code>
<code>kSpritePropertyVisible</code>	<code>short *</code>
<code>kSpritePropertyLayer</code>	<code>short *</code>
<code>kSpritePropertyGraphicsMode</code>	<code>ModifierTrackGraphicsModeRecord *</code>
<code>kSpritePropertyImageIndex</code>	<code>short *</code>

RESULT CODES

<code>noErr</code>	<code>0</code>	No error
<code>invalidSpritePropertyErr</code>	<code>-2065</code>	Specified sprite property does not exist
<code>invalidSpriteIndexErr</code>	<code>-2067</code>	Sprite index is out of range

SpriteMediaHitTestAllSprites

The `SpriteMediaHitTestAllSprites` function determines whether any sprites are at a specified location.

```
pascal ComponentResult SpriteMediaHitTestAllSprites (  
    MediaHandler mh,  
    long flags,  
    Point loc,  
    QTAAtomID *spriteHitID);
```

Sprite Media Handler API Reference

<code>mh</code>	Specifies the sprite media handler for this operation.
<code>flags</code>	Specifies flags to control the hit testing operation. The following flags are valid for use with <code>SpriteMediaHitTestAllSprites</code> and <code>SpriteMediaHitTestOneSprite</code> : <ul style="list-style-type: none"> <code>spriteHitTestBounds</code> The specified location must be within the sprite's bounding box. <code>spriteHitTestImage</code> If both this flag and <code>spriteHitTestBounds</code> are set, the specified location must be within the shape of the sprite's image. <code>spriteHitTestInvisibleSprites</code> Set this flag if you want invisible sprites to be hit tested along with the visible ones. <code>spriteHitTestLocInDisplayCoordinates</code> Set this flag if the hit testing point is in display coordinates instead of local sprite track coordinates. <code>spriteHitTestIsClick</code> Set this flag if you want the hit testing operation to pass a click on to the codec currently rendering the sprites image. For example, this can be used to make the Ripple codec ripple.
<code>loc</code>	Specifies a point in the coordinate system of the sprite track's movie to test for the existence of a sprite.
<code>spriteHitID</code>	Contains a pointer to a <code>short</code> integer. On return, this integer contains the ID of the frontmost sprite at the location specified by <code>loc</code> . If no sprite exists at the location, the function sets the value of this parameter to 0.

DISCUSSION

You call this function to determine whether any sprites exist at a specified location in the coordinate system of a sprite track's movie. You can pass flags to

this function to control the hit testing operation more precisely. For example, you may want the hit test operation to detect a sprite whose bounding box contains the specified location.

SpriteMediaCountSprites

The `SpriteMediaCountSprites` function retrieves the number of sprites that currently exist in a sprite track.

```
pascal ComponentResult SpriteMediaCountSprites (
    MediaHandler mh,
    short* numSprites);
```

<code>mh</code>	Specifies the sprite media handler for this operation.
<code>numSprites</code>	Contains a pointer to a <code>short</code> integer. On return, this integer contains the number of sprites for the sprite media's current time.

DISCUSSION

This function determines the number of sprites that currently exist based on the key frame that is in effect.

SpriteMediaCountImages

The `SpriteMediaCountImages` function retrieves the number of images that currently exist in a sprite track.

```
pascal ComponentResult SpriteMediaCountImages (
    MediaHandler mh,
    short* numImages);
```

<code>mh</code>	Specifies the sprite media handler for this operation.
-----------------	--

`numImages` Contains a pointer to a `short` integer. On return, this integer contains the number of images for the sprite media's current time.

DISCUSSION

This function determines the number of images that currently exist based on the key frame that is in effect.

SpriteMediaGetIndImageDescription

The `SpriteMediaGetIndImageDescription` function retrieves an image description for the specified image in a sprite track.

```
pascal ComponentResult SpriteMediaGetIndImageDescription (
    MediaHandler mh,
    short imageIndex,
    ImageDescriptionHandle imageDescription);
```

`mh` Specifies the sprite media handler for this operation.

`imageIndex` Specifies the index of the image whose image description should be retrieved.

`imageDescription` Specifies an image description handle. On return, this handle contains the image description for the specified image.

DISCUSSION

You set the `imageIndex` parameter to the index of the image whose image description you want to retrieve. The index must be between one and the number of available images. You can determine how many images are available by calling `SpriteMediaCountImages`.

The handle specified by the `imageDescription` parameter must be unlocked; this function resizes the handle if necessary.

RESULT CODES

<code>noErr</code>	0	No error
<code>paramErr</code>	-50	Invalid parameter specified
<code>invalidImageIndexErr</code>	-2068	Image index is out of range

Memory Manager errors, as documented in *Mac OS For QuickTime Programmers*.

SpriteMediaGetDisplayedSampleNumber

The `SpriteMediaGetDisplayedSampleNumber` function retrieves the number of the sample that is currently being displayed.

```
pascal ComponentResult SpriteMediaGetDisplayedSampleNumber (
    MediaHandler mh,
    long* sampleNum);
```

<code>mh</code>	Specifies the sprite media handler for this operation.
<code>sampleNum</code>	Contains a pointer to a <code>long</code> integer. On return, this integer contains the number of the sample that is currently being displayed.

DISCUSSION

You call this function when you need to retrieve the sample number of the sample that is being displayed.

SpriteMediaGetSpriteName

The `SpriteMediaGetSpriteName` function returns the name of the sprite with the specified ID from the currently displayed sample.

```
pascal ComponentResult SpriteMediaGetSpriteName(
    MediaHandler mh,
    QTAtomID spriteID,
    Str255 spriteName);
```

<code>mh</code>	Specifies the sprite media handler for this operation.
-----------------	--

<code>spriteID</code>	Specifies the sprite ID of the sprite name.
<code>spriteName</code>	Returns a Pascal string with the name of the sprite or an empty string if the sprite is unnamed.

SpriteMediaGetImageName

The `SpriteMediaGetImageName` function returns the name of the image with the specified index from the current key frame sample.

```
pascal ComponentResult SpriteMediaGetImageName(
    MediaHandler mh,
    short imageIndex,
    Str255 imageName);
```

<code>mh</code>	Specifies the sprite media handler for this operation.
<code>imageIndex</code>	Specifies the index of the image whose image name should be retrieved.
<code>imageName</code>	Returns a Pascal string with the image name of the image or an empty string if the image is unnamed.

SpriteMediaHitTestOneSprite

The `SpriteMediaHitTestOneSprite` function performs a hit testing operation on the sprite specified by the `spriteID`.

```
pascal ComponentResult SpriteMediaHitTestOneSprite(
    MediaHandler mh,
    QTAAtomID spriteID,
    long flags,
    Point loc,
    Boolean *wasHit);
```

<code>mh</code>	Specifies the sprite media handler for this operation.
<code>spriteID</code>	Specifies the sprite ID of the sprite name.

Sprite Media Handler API Reference

<code>flags</code>	Specifies the flags to control the hit testing operation.
<code>loc</code>	The location <code>loc</code> should be in local coordinates of the sprite track, unless the <code>spriteHitTestLocInDisplayCoordinates</code> flag is set.
<code>wasHit</code>	Contains a pointer to a Boolean. If the sprite is hit, <code>wasHit</code> is set to <code>true</code> ; otherwise, it is set to <code>false</code> . This routine allows you to hit test a sprite which is fully or partially covered by other sprites.

Note

Sprite indexes range from 1 to the number of sprites in the key sample. ♦

SpriteMediaSpriteIndexToID

The `SpriteMediaSpriteIndexToID` function returns the ID of the sprite specified by `spriteIndex` in `spriteID`.

```
pascal ComponentResult SpriteMediaSpriteIndexToID(
    MediaHandler mh,
    short spriteIndex,
    QAtomID *spriteID);
```

<code>mh</code>	Specifies the sprite media handler for this operation.
<code>spriteIndex</code>	Specifies the index of the sprite for this operation.
<code>spriteID</code>	Contains a pointer to the sprite ID of the sprite index. If a sprite with the specified index does not exist, the error <code>paramErr</code> is returned.

SpriteMediaIDToIndex

The `SpriteMediaSpriteIDToIndex` function returns the sprite index of the sprite specified by the `spriteID`.

```
pascal ComponentResult SpriteMediaSpriteIDToIndex(
    MediaHandler mh,
    QTAtomID spriteID,
    short *spriteIndex);
```

mh Specifies the sprite media handler for this operation.

spriteID Specifies the sprite ID of the sprite index.

spriteIndex Contains a pointer to a `short` integer. If a sprite with the specified ID does not exist, the error `invalidSpriteIDErr` is returned.

SpriteMediaGetIndImageProperty

The `SpriteMediaGetIndImageProperty` function returns a property value for the image specified by `imageIndex`.

```
pascal ComponentResult SpriteMediaGetIndImageProperty(
    MediaHandler mh,
    short imageIndex,
    long imagePropertyType,
    void *imagePropertyValue);
```

mh Specifies the sprite media handler for this operation.

imageIndex Specifies the index of the image.

imagePropertyType Specifies an image property type whose value is returned in `imagePropertyValue`. The allowed property types are:

`kSpriteImagePropertyRegistrationPoint`
`imagePropertyValue` is a `FixedPoint` value.

`kSpriteImagePropertyGroupID`
`imagePropertyValue` is a long integer.

imagePropertyValue

A pointer is set to the value of the image property.

Sprites Functions Specific to Wired Sprites

The following routines are specific to sprite tracks using wired sprites. For more information on wired sprites, see Chapter 1, “Introduction to Wired Movies, Sprites, and the Sprite Toolbox.”

SpriteMediaSetActionVariable

The `SpriteMediaSetActionVariable` function sets the value of the sprite track variable with the ID of the variable to the supplied value.

```
pascal ComponentResult SpriteMediaSetActionVariable
    (MediaHandler mh,
     QTAtomID variableID,
     const float *value);
```

<code>mh</code>	Specifies the sprite media handler for this operation.
<code>variableID</code>	Specifies a variable ID of the sprite name.
<code>value</code>	Contains a pointer to a floating-point number. Note that the value is passed by reference.

SpriteMediaGetActionVariable

The `SpriteMediaGetActionVariable` returns the value of the sprite track variable with the specified ID.

```
pascal ComponentResult SpriteMediaGetActionVariable(
    MediaHandler mh,
    QTAtomID variableID,
    float *value);
```

<code>mh</code>	Specifies the sprite media handler for this operation.
-----------------	--

<code>variableID</code>	Specifies a variable ID of the sprite variable.
<code>value</code>	Contains a pointer to a floating-point value. If the specified variable has never been set, the value is set to 0 and the error <code>cannotFindAtomErr</code> is returned.

SpriteDescription Structure

Sprite samples may be compressed using a data compression codec.

```
struct SpriteDescription {
    long        descSize;           /* total size of
                                   SpriteDescription including extra data */
    long        dataFormat;         /* */
    long        resvd1;             /* reserved for apple use */
    short       resvd2;
    short       dataRefIndex;
    long        version;            /* which version is this data */
    OSType      decompressorType;   /* which decompressor to use, 0 for
                                   no decompression */
    long        sampleFlags;        /* how to interpret samples */
};
typedef struct SpriteDescription    SpriteDescription;
typedef SpriteDescription *        SpriteDescriptionPtr;
typedef SpriteDescriptionPtr *     SpriteDescriptionHandle;
```

Field descriptions

<code>decompressorType</code>	This field in the <code>SpriteDescription</code> sample description structure allows a data decompressor component type to be specified. If this field is nonzero, a component of the specified type is used to decompress the sprite sample when it is loaded.
-------------------------------	---

QTAtomContainer-Based Data Structure Descriptions

QTAtomContainer-based data structures are being widely used in QuickTime. This appendix is an attempt at standardizing how the format of these data structures may be described and is documented. The key presented here is used in the chapters on “Tween Media Handler” and “Tween Components and Native Tween Types” in *QuickTime 4 Reference*.

QTAtomContainer Description Key

```
[(QTAtomFormatName)] =
    atomType_1, id, index
    data
    atomType_n, id, index
    data
```

The atoms may be required or optional:

```
<atomType>  optional atom
atomType    required atom
```

The atom id may be a number if it is required to be a constant, or may be a list of valid atom id's, indicating that multiple atoms of this type are allowed.

```
3                one atom with id of 3
(1..3)           three atoms with id's of 1, 2, and 3
(1, 5, 7)        three atoms with id's of 1, 5, and 7
(anyUniqueIDs)   multiple atoms each with a unique id
```

The atom index may be a 1 if only one atom of this type is allowed, or it may be a range from one to some constant or variable.

APPENDIX

QTAtomContainer-Based Data Structure Descriptions

1 one atom of this type is allowed, index is always 1
(1..3) three atoms with indices 1, 2, and 3
(1..numAtoms) numAtoms atoms with indices of 1 to numAtoms

The data may be leaf data in which its data type is listed inside of brackets [], or may be a nested tree of atoms.

[theDataType] leaf data of type theDataType
childAtoms a nested tree of atoms

Nested QTAtom format definitions [(AtomFormatName)] may appear in a definition.

Sprite Media Handler Media Format Definition

This appendix contains the following:

- The sprite media handler sample description structure
- The sprite media handler track properties QTAtomContainer format
- The sprite media handler sample QTAtomContainer formats

Sprite MediaHandler Sample Description Structure

```

struct SpriteDescription {
    long        descSize;           /* total size of
SpriteDescription
                                   including extra data */
    long        dataFormat;         /* */
    long        resvd1;             /* reserved for apple use */
    short       resvd2;
    short       dataRefIndex;
    long        version;            /* which version is this data */
    OSType      decompressorType;  /* which decompressor to use,
                                   0 for no decompression */
    long        sampleFlags;        /* how to interpret samples */
};
typedef struct SpriteDescription SpriteDescription;
typedef SpriteDescription *      SpriteDescriptionPtr;
typedef SpriteDescriptionPtr *   SpriteDescriptionHandle;

```

Sprite MediaHandler Track Properties QTAtomContainer Format

```
[(SpriteTrackProperties)]
    <kSpriteTrackPropertyBackgroundColor, 1, 1>
        [RGBColor]
    <kSpriteTrackPropertyOffscreenBitDepth, 1, 1>
        [short]
    <kSpriteTrackPropertySampleFormat, 1, 1>
        [long]
    <kSpriteTrackPropertyScaleSpritesToScaleWorld, 1, 1>
        [Boolean]
    <kSpriteTrackPropertyHasActions, 1, 1>
        [Boolean]
    <kSpriteTrackPropertyVisible, 1, 1>
        [Boolean]
    <kSpriteTrackPropertyQTIdleEventsFrequency, 1, 1>
        [UInt32]
```

Sprite MediaHandler Sample QTAtomContainer Formats

```
[(SpriteKeySample)] =
    [(SpritePropertyAtoms)]
    [(SpriteImageAtoms)]

[(SpriteOverrideSample)] =
    [(SpritePropertyAtoms)]

[(SpriteImageAtoms)]
    kSpriteSharedDataAtomType, 1, 1
    <kSpriteVariablesContainerAtomType>, 1
        <kSpriteStringVariableAtomType>, (1..n) ID is SpriteTrack
            Variable ID to be set
                [CString]
        <kSpriteFloatingPointVariableAtomType>, (1..n) ID is
            SpriteTrack Variable ID to be set
                [float]
```


APPENDIX B

Sprite Media Handler Media Format Definition

```
kSpriteImagesContainerAtomType, 1, 1
    kSpriteImageAtomType, theImageID, (1 .. numImages)
        kSpriteImageDataAtomType, 1, 1
            [ImageData is ImageDescriptionHandle prepended to
                                                    image data]

        <kSpriteImageRegistrationAtomType, 1, 1>
            [FixedPoint]
        <kSpriteImageNameAtomType, 1, 1>
            [pString]
        <kSpriteImageGroupIDAtomType, 1, 1>
            [long]

[(SpritePropertyAtoms)]
    <kQTEventFrameLoaded>, 1, 1
        [(ActionListAtoms)]
        <kCommentAtomType>, (anyUniqueIDs), (1..numComments)
            [CString]

kSpriteAtomType, theSpriteID, (1 .. numSprites)
    <kSpritePropertyMatrix, 1, 1>
        [MatrixRecord]
    <kSpritePropertyVisible, 1, 1>
        [short]
    <kSpritePropertyLayer, 1, 1>
        [short]
    <kSpritePropertyImageIndex, 1, 1>
        [short]
    <kSpritePropertyGraphicsMode, 1, 1>
        [ModifierTrackGraphicsModeRecord]

    <kSpriteUsesImageIDsAtomType, 1, 1>
        [array of QTAtomID's, one per image used]

    <kSpriteBehaviorsAtomType>, 1

    <kSpriteImageBehaviorAtomType>
        [QTSpriteButtonBehaviorStruct]
    <kSpriteCursorBehaviorAtomType>
        [QTSpriteButtonBehaviorStruct]
```

APPENDIX B

Sprite Media Handler Media Format Definition

```
<kSpriteStatusStringsBehaviorAtomType>
    [QTSpriteButtonBehaviorStruct]

<[(SpriteActionAtoms)]>

[(SpriteActionAtoms)] =
    kQTEventType, theQTEventType, (1 .. numEventTypes)
    [(ActionListAtoms)]
    <kCommentAtomType>, (anyUniqueIDs), (1..numComments)
    [CString]

[(ActionListAtoms)] =
    kAction, (anyUniqueIDs), (1..numActions)
    kWhichAction    1, 1
    [long whichActionConstant]
    <kActionParameter> (anyUniqueIDs), (1..numParameters)
    [(parameterData)] ( whichActionConstant, paramIndex )
    // either leaf data or child atoms
    <kActionFlags> parameterID, (1..numParamsWithFlags)
    [long actionFlags]
    <kActionParameterMinValue> parameterID, (1.. numParamsWithMin)
    [data depends on param type]
    <kActionParameterMaxValue> parameterID, (1.. numParamsWithMax)
    [data depends on param type]
    [(ActionTargetAtoms)]

    <kCommentAtomType>, (anyUniqueIDs), (1..numComments)
    [CString]

[(ActionTargetAtoms)] =
    <kActionTarget>
    <kTargetMovie>
    [no data]
[(ActionTargetAtoms)] =
    <kActionTarget>

    <kTargetMovieName>
    [Pstring MovieName]
```

APPENDIX B

Sprite Media Handler Media Format Definition

```
OR
<kTargetMovieID>
    [long MovieID]
OR
    [(kExpressionAtoms)]
<kTargetTrackName>
    [PString trackName]
<kTargetTrackType>
    [OSType trackType]
<kTargetTrackIndex>
    [long trackIndex]
OR
    [(kExpressionAtoms)]
<kTargetTrackID>
    [long trackID]
OR
    [(kExpressionAtoms)]
<kTargetSpriteName>
    [PString spriteName]
<kTargetSpriteIndex>
    [short spriteIndex]
OR
    [(kExpressionAtoms)]
<kTargetSpriteID>
    [QTAtomID spriteIID]
OR
    [(kExpressionAtoms)]

[(kExpressionAtoms)] =
    kExpressionContainerAtomType, 1, 1
    <kOperatorAtomType, theOperatorType, 1>
        kOperandAtomType, (anyUniqueIDs), (1..numOperands)
        [(OperandAtoms)]
OR
    <kOperandAtomType, 1, 1>
        [(OperandAtoms)]

[(OperandAtoms)] =
```

APPENDIX B

Sprite Media Handler Media Format Definition

```
<kOperandExpression> 1, 1
    [(kExpressionAtoms)]                // allows for recursion
OR
<kOperandConstant> 1, 1
    [ float theConstant ]
OR
<kOperandSpriteTrackVariable> 1, 1
    [(ActionTargetAtoms)]
    kActionParameter, 1, 1
    [QAtomID spriteVariableID]
OR
<kOperandKeyIsDown> 1, 1
    kActionParameter, 1, 1
    [UInt16 modifierKeys]
    kActionParameter, 2, 2
    [UInt8 asciiCharCode]
OR
<kOperandRandom> 1, 1
    kActionParameter, 1, 1
    [short minimum]
    kActionParameter, 2, 2
    [short maximum]
OR
<any other operand atom type>
    [(ActionTargetAtoms)]
```

The format for parameter data depends on the action and parameter index.

In most cases, the `kActionParameter` atom is a leaf atom containing data; for a few parameters, it contains child atoms.

`whichAction` corresponds to the action type which is specified by the leaf data of a `kWhichAction` atom.

`paramIndex` is the index of the parameter's `kActionParameter` atom.

```
[(parameterData)] ( whichAction, paramIndex ) =
{
    kActionMovieSetVolume:
        param1:      short volume

    kActionMovieSetRate
        param1:      Fixed rate
```

APPENDIX B

Sprite Media Handler Media Format Definition

```
kActionMovieSetLoopingFlags
    param1:    long loopingFlags

kActionMovieGoToTime
    param1:    TimeValue time

kActionMovieGoToTimeByName
    param1:    Str255 timeName

kActionMovieGoToBeginning
    no params

kActionMovieGoToEnd
    no params

kActionMovieStepForward
    no params

kActionMovieStepBackward
    no params

kActionMovieSetSelection
    param1:    TimeValue startTime
    param2:    TimeValue endTime

kActionMovieSetSelectionByName
    param1:    Str255 startTimeName
    param2:    Str255 endTimeName

kActionMoviePlaySelection
    param1:    Boolean selectionOnly

kActionMovieSetLanguage
    param1:    long language

kActionMovieChanged
    no params

kActionTrackSetVolume
```

APPENDIX B

Sprite Media Handler Media Format Definition

```
    param1:    short volume

kActionTrackSetBalance
    param1:    short balance

kActionTrackSetEnabled
    param1:    Boolean enabled

kActionTrackSetMatrix
    param1:    MatrixRecord matrix

kActionTrackSetLayer
    param1:    short layer

kActionTrackSetClip
    param1:    RgnHandle clip


kActionSpriteSetMatrix
    param1:    MatrixRecord matrix

kActionSpriteSetImageIndex
    param1:    short imageIndex

kActionSpriteSetVisible
    param1:    short visible

kActionSpriteSetLayer
    param1:    short layer

kActionSpriteSetGraphicsMode
    param1:    ModifierTrackGraphicsModeRecord graphicsMode

kActionSpritePassMouseToCodec
    no params

kActionSpriteClickOnCodec
    param1:    Point localLoc

kActionSpriteTranslate
```

APPENDIX B

Sprite Media Handler Media Format Definition

```
param1:    Fixed x
param2:    Fixed y
param3:    Boolean isRelative
```

kActionSpriteScale

```
param1:    Fixed xScale
param2:    Fixed yScale
```

kActionSpriteRotate

```
param1:    Fixed degrees
```

kActionSpriteStretch

```
param1:    Fixed p1x
param2:    Fixed p1y
param3:    Fixed p2x
param4:    Fixed p2y
param5:    Fixed p3x
param6:    Fixed p3y
param7:    Fixed p4x
param8:    Fixed p4y
```

kActionQTVRSetPanAngle

```
param1:    float panAngle
```

kActionQTVRSetTiltAngle

```
param1:    float tileAngle
```

kActionQTVRSetFieldOfView

```
param1:    float fieldOfView
```

kActionQTVRShowDefaultView

```
no params
```

kActionQTVRGoToNodeID

```
param1:    UInt32 nodeID
```

kActionMusicPlayNote

```
param1:    long sampleDescIndex
param2:    long partNumber
param3:    long delay
```

APPENDIX B

Sprite Media Handler Media Format Definition

```
param4:    long pitch
param5:    long velocity
param6:    long duration
```

kActionMusicSetController

```
param1:    long sampleDescIndex
param2:    long partNumber
param3:    long delay
param4:    long controller
param5:    long value
```

kActionCode

```
param1:    [(CaseStatementActionAtoms)]
```

kActionWhile

```
param1:    [(WhileStatementActionAtoms)]
```

kActionGoToURL

```
param1:    CString urlLink
```

kActionSendQTEventToSprite

```
param1:    [(SpriteTargetAtoms)]
param2:    QTEventRecord theEvent
```

kActionDebugStr

```
param1:    Str255 theMessageString
```

kActionPushCurrentTime

```
no params
```

kActionPushCurrentTimeWithLabel

```
param1:    Str255 theLabel
```

kActionPopAndGotoTopTime

```
no params
```

kActionPopAndGotoLabeledTime

```
param1:    Str255 theLabel
```

kActionSpriteTrackSetVariable

```
param1:    QTAtomID variableID
```


APPENDIX B

Sprite Media Handler Media Format Definition

```
        param2:      float value

    kActionApplicationNumberAndString
        param1:      long aNumber
        param2:      Str255 aString
}
```

Both [(CaseStatementActionAtoms)] and [(WhileStatementActionAtoms)] are child atoms of a kActionParameter 1, 1 atom

```
[(CaseStatementActionAtoms)] =
    kConditionalAtomType, (anyUniqueIDs), (1..numCases)
    [(kExpressionAtoms)]
    kActionListAtomType 1, 1
    [(ActionListAtoms)]          // may contain nested conditional
                                   actions
```

```
[(WhileStatementActionAtoms)] =
    kConditionalAtomType, 1, 1
    [(kExpressionAtoms)]
    kActionListAtomType 1, 1
    [(ActionListAtoms)]          // may contain nested
                                   conditional actions
```

A P P E N D I X B

Sprite Media Handler Media Format Definition

QTWiredSprite.c Sample Code

This appendix includes sample code from `QTWiredSprite.c`. Note this is only a partial code listing. You can download the full sample code at QuickTime Web site at <http://www.apple.com/quicktime/developers/samplecode.html#sprites>.) It is also available on the QuickTime SDK.

For information on how to use this sample code in your application, see Chapter 2, “The Sprite Media Handler,” and Chapter 3, “Authoring Wired Movies and Sprite Animations.”

```

//////////
//
//  File:          QTWiredSprites.c
//
//  Contains:      QuickTime wired sprites support for QuickTime movies.
//

//  Written by:   Sean Allen
//  Revised by:   Chris Flick and Tim Monroe
//               Based (heavily!) on the existing MakeActionSpriteMovie.c
//               code written by Sean Allen.
//
//  Copyright:    © 1997-1999 by Apple Computer, Inc., all rights reserved.
//
//  Change History (most recent first):
//
//      <2>        03/26/98    rtm        made fixes for Windows compiles
//      <1>        03/25/98    rtm        first file; integrated existing code
//                                       with shell framework
//
//
//  This sample code creates a wired sprite movie containing one sprite
//  track. The sprite track contains six sprites: two penguins and four
//  buttons.
//
//  The four buttons are initially invisible. When the mouse enters (or
//  "rolls over") a button, it appears.

```

APPENDIX C

QTWiredSprite.c Sample Code

```
// When the mouse is clicked inside a button, its image changes to its /
// "pressed" image. When the mouse
// is released, its image changes back to its "unpressed" image. If the
// mouse is released inside the button,
// an action is triggered. The buttons perform the actions of go to
// beginning of movie, step backward,
// step forward, and go to end of movie.

//
// The first penguin shows all of the buttons when the mouse enters it,
// and hides them when the mouse exits.
// The first penguin is the only sprite that has properties that are
// overridden by the override sprite samples.
// These samples override its matrix (in order to move it) and its image
// index (in order to make it "waddle").
//
// When the mouse is clicked on the second penguin, it changes its image
// index to its "eyes closed" image.
// When the mouse is released, it changes back to its normal image. This
// makes it appear to blink when clicked on.
// When the mouse is released over the penguin, several actions are
// triggered. Both penguins' graphics states are
// toggled between copyMode and blendMode, and the movie's rate is
// toggled between zero and one.
//
// The second penguin moves once per second. This occurs whether the
// movie's rate is currently zero or one,
// because it is being triggered by a gated idle event. When the penguin
// receives the idle event, it changes
// its matrix using an action which uses min, max, delta, and wraparound
// options.
//
// The movie's looping mode is set to palindrome by a frame-loaded
// action.
//
// So, our general strategy is as follows (though perhaps not in the
// order listed):
//
// (1) Create a new movie file with a single sprite track.
// (2) Assign the "no controller" movie controller to the movie.
// (3) Set the sprite track's background color, idle event
```

APPENDIX C

QTWiredSprite.c Sample Code

```
//      frequency, and hasActions properties.
//      (4) Convert our PICT resources to animation codec images with
//      transparency.
//      (5) Create a key frame sample containing six sprites and all of
//      their shared images.
//      (6) Assign the sprites their initial property values.
//      (7) Create a frameLoaded event for the key frame.
//      (8) Create some override samples that override the matrix and
//      image index properties of the first penguin sprite.
//

//  NOTES:
//
//  *** (1) ***
//  There are event types other than mouse related events (for instance,
//  Idle and FrameLoaded).
//  Idle events are independent of the movie's rate, and they can be
//  gated so they are sent at most
//  every n ticks. In our sample movie, the second penguin moves when the
//  movie's rate is zero,
//  and moves only once per second because of the value of the sprite
//  track's idleEventFrequency property.
//
//  *** (2) ***
//  Multiple actions may be executed in response to a single event. In
//  our sample movie, rolling over
//  the first penguin shows and hides four different buttons.
//
//  *** (3) ***
//  Actions may target any sprite or track in the movie. In our sample
//  movie, clicking on one penguin
//  changes the graphics mode of the other.
//
//  *** (4) ***
//  Conditional and looping control structures are supported. In our
//  sample movie, the second penguin
//  uses the "case statement" action.
//
//  *** (5) ***
//  Sprite track variables that have not been set have a default value of
//  zero. (The second penguin's
```

APPENDIX C

QTWiredSprite.c Sample Code

```
// conditional code relies on this.)
//
// *** (6) ***
// Wired sprites were previously known as "action sprites". Don't let
// the names of some of the utility
// functions confuse you. We'll try to update the source code as time
// permits.
//
// *** (7) ***
// Penguins don't fly, but I hear they totally shred halfpipes on
// snowboards.
//
//////////
// header files
#include "QTWiredSprites.h"

//////////
//
// QTWired_CreateWiredSpritesMovie
// Create a QuickTime movie containing a wired sprites track.
//
//////////

OSErr QTWired_CreateWiredSpritesMovie (void)
{
    short                myResRefNum = 0;
    Movie                myMovie = NULL;
    Track                myTrack;
    Media                myMedia;
    StandardFileReply    myReply;
    QTAtomContainer      mySample = NULL;
    QTAtomContainer      myActions = NULL;
    QTAtomContainer      myBeginButton, myPrevButton, myNextButton,
                        myEndButton;
    QTAtomContainer      myPenguinOne, myPenguinTwo,
                        myPenguinOneOverride;
    QTAtomContainer      myBeginActionButton, myPrevActionButton,
                        myNextActionButton, myEndActionButton;
    QTAtomContainer      myPenguinOneAction, myPenguinTwoAction;
    RGBColor             myKeyColor;
```

APPENDIX C

QTWiredSprite.c Sample Code

```
Point                myLocation;
short               isVisible, myLayer, myIndex, myResID, i,
                   myDelta;
Boolean             hasActions;
long               myFlags = createMovieFileDeleteCurFile |
                           createMovieFileDontCreateResFile;
OSType              myType = FOUR_CHAR_CODE('none');
UInt32              myFrequency;
QTAtom              myEventAtom;
long               myLoopingFlags;
ModifierTrackGraphicsModeRecord myGraphicsMode;
OSErr               myErr = noErr;

//////////
//
// create a new movie file and set itscontroller type
//
//////////

// ask the user for the name of the new movie file
StandardPutFile("\pSprite movie file name:", "\pSprite.mov",
               &myReply);
if (!myReply.sfGood)
    goto bail;

// create a movie file for the destination movie
myErr = CreateMovieFile(&myReply.sfFile, FOUR_CHAR_CODE('TVOD'), 0,
                      myFlags, &myResRefNum, &myMovie);

if (myErr != noErr)
    goto bail;

// select the "no controller" movie controller
myType = EndianU32_NtoB(myType);
SetUserDataItem(GetMovieUserData(myMovie), &myType, sizeof(myType),
               kUserDataMovieControllerType, 1);

//////////
//
// create the sprite track and media
//
//////////
```

QTWiredSprite.c Sample Code

```

myTrack = NewMovieTrack(myMovie, ((long)kSpriteTrackWidth << 16),
                        ((long)kSpriteTrackHeight << 16), kNoVolume);
myMedia = NewTrackMedia(myTrack, SpriteMediaType,
kSpriteMediaTimeScale, NULL, 0);

//////////
//
// create a key frame sample containing six sprites and all of their
// shared images
//
//////////

// create a new, empty key frame sample
myErr = QTNewAtomContainer(&mySample);
if (myErr != noErr)
    goto bail;

myKeyColor.red = 0xffff;                // white
myKeyColor.green = 0xffff;
myKeyColor.blue = 0xffff;

// add images to the key frame sample
AddPictImageToKeyFrameSample(mySample, kGoToBeginningButtonUp,
                             &myKeyColor, kGoToBeginningButtonUpIndex, NULL, NULL);
AddPictImageToKeyFrameSample(mySample, kGoToBeginningButtonDown,
                             &myKeyColor, kGoToBeginningButtonDownIndex, NULL, NULL);
AddPictImageToKeyFrameSample(mySample, kGoToEndButtonUp, &myKeyColor,
                             kGoToEndButtonUpIndex, NULL, NULL);
AddPictImageToKeyFrameSample(mySample, kGoToEndButtonDown,
                             &myKeyColor, kGoToEndButtonDownIndex, NULL, NULL);
AddPictImageToKeyFrameSample(mySample, kGoToPrevButtonUp,
                             &myKeyColor, kGoToPrevButtonUpIndex, NULL, NULL);
AddPictImageToKeyFrameSample(mySample, kGoToPrevButtonDown,
                             &myKeyColor, kGoToPrevButtonDownIndex, NULL, NULL);
AddPictImageToKeyFrameSample(mySample, kGoToNextButtonUp,
                             &myKeyColor, kGoToNextButtonUpIndex, NULL, NULL);
AddPictImageToKeyFrameSample(mySample, kGoToNextButtonDown,
                             &myKeyColor, kGoToNextButtonDownIndex, NULL, NULL);
AddPictImageToKeyFrameSample(mySample, kPenguinForward, &myKeyColor,
                             kPenguinForwardIndex, NULL, NULL);

```


APPENDIX C

QTWiredSprite.c Sample Code

```
AddPictImageToKeyFrameSample(mySample, kPenguinLeft, &myKeyColor,
                              kPenguinLeftIndex, NULL, NULL);
AddPictImageToKeyFrameSample(mySample, kPenguinRight, &myKeyColor,
                              kPenguinRightIndex, NULL, NULL);
AddPictImageToKeyFrameSample(mySample, kPenguinClosed, &myKeyColor,
                              kPenguinClosedIndex, NULL, NULL);

for (myIndex = kPenguinDownRightCycleStartIndex, myResID =
kWalkDownRightCycleStart; myIndex <= kPenguinDownRightCycleEndIndex;
myIndex++, myResID++)
    AddPictImageToKeyFrameSample(mySample, myResID, &myKeyColor,
                                myIndex, NULL, NULL);

// assign group IDs to the images
AssignImageGroupIDsToKeyFrame(mySample);

//////////
//
// add samples to the sprite track's media
//
//////////

BeginMediaEdits(myMedia);

// go to beginning button with no actions
myErr = QTNewAtomContainer(&myBeginButton);
if (myErr != noErr)
    goto bail;
myLocation.h    = (1 * kSpriteTrackWidth / 8) -
                  (kStartEndButtonWidth / 2);
myLocation.v    = (4 * kSpriteTrackHeight / 5) -
                  (kStartEndButtonHeight / 2);
isVisible       = false;
myLayer         = 1;
myIndex         = kGoToBeginningButtonUpIndex;
myErr = SetSpriteData(myBeginButton, &myLocation, &isVisible,
                      &myLayer, &myIndex, NULL, NULL, myActions);
if (myErr != noErr)
    goto bail;

// go to previous button with no actions
```

APPENDIX C

QTWiredSprite.c Sample Code

```
myErr = QTNewAtomContainer(&myPrevButton);
if (myErr != noErr)
    goto bail;
myLocation.h    = (3 * kSpriteTrackWidth / 8) -
                  (kNextPrevButtonWidth / 2);
myLocation.v    = (4 * kSpriteTrackHeight / 5) -
                  (kStartEndButtonHeight / 2);
isVisible       = false;
myLayer         = 1;
myIndex         = kGoToPrevButtonUpIndex;
myErr = SetSpriteData(myPrevButton, &myLocation, &isVisible,
                      &myLayer, &myIndex, NULL, NULL, myActions);

if (myErr != noErr)
    goto bail;

// go to next button with no actions
myErr = QTNewAtomContainer(&myNextButton);
if (myErr != noErr)
    goto bail;
myLocation.h    = (5 * kSpriteTrackWidth / 8) -
                  (kNextPrevButtonWidth / 2);
myLocation.v    = (4 * kSpriteTrackHeight / 5) -
                  (kStartEndButtonHeight / 2);
isVisible       = false;
myLayer         = 1;
myIndex         = kGoToNextButtonUpIndex;
myErr = SetSpriteData(myNextButton, &myLocation, &isVisible,
                      &myLayer, &myIndex, NULL, NULL, myActions);

if (myErr != noErr)
    goto bail;

// go to end button with no actions
myErr = QTNewAtomContainer(&myEndButton);
if (myErr != noErr)
    goto bail;
myLocation.h    = (7 * kSpriteTrackWidth / 8) -
                  (kStartEndButtonWidth / 2);
myLocation.v    = (4 * kSpriteTrackHeight / 5) -
                  (kStartEndButtonHeight / 2);
isVisible       = false;
myLayer         = 1;
```

QTWiredSprite.c Sample Code

```

myIndex          = kGoToEndButtonUpIndex;
myErr = SetSpriteData(myEndButton, &myLocation, &isVisible, &myLayer,
                      &myIndex, NULL, NULL, myActions);

if (myErr != noErr)
    goto bail;

// first penguin sprite with no actions
myErr = QTNewAtomContainer(&myPenguinOne);
if (myErr != noErr)
    goto bail;
myLocation.h     = (3 * kSpriteTrackWidth / 8) - (kPenguinWidth / 2);
myLocation.v     = (kSpriteTrackHeight / 4) - (kPenguinHeight / 2);
isVisible        = true;
myLayer          = 2;
myIndex          = kPenguinDownRightCycleStartIndex;
myGraphicsMode.graphicsMode = blend;
myGraphicsMode.opColor.red = myGraphicsMode.opColor.green =
myGraphicsMode.opColor.blue = 0x8FFF;           // grey
myErr = SetSpriteData(myPenguinOne, &myLocation, &isVisible,
                      &myLayer, &myIndex, &myGraphicsMode, NULL, myActions);
if (myErr != noErr)
    goto bail;

// second penguin sprite with no actions
myErr = QTNewAtomContainer(&myPenguinTwo);
if (myErr != noErr)
    goto bail;
myLocation.h     = (5 * kSpriteTrackWidth / 8) - (kPenguinWidth / 2);
myLocation.v     = (kSpriteTrackHeight / 4) - (kPenguinHeight / 2);
isVisible        = true;
myLayer          = 3;
myIndex          = kPenguinForwardIndex;
myErr = SetSpriteData(myPenguinTwo, &myLocation, &isVisible,
                      &myLayer, &myIndex, NULL, NULL, myActions);
if (myErr != noErr)
    goto bail;

//////////
//
// add actions to the six sprites
//

```

APPENDIX C

QTWiredSprite.c Sample Code

```
//////////

// add go to beginning button
myErr = QTCopyAtom(myBeginButton, kParentAtomIsContainer,
&myBeginActionButton);
if (myErr != noErr)
    goto bail;

AddSpriteSetImageIndexAction(myBeginActionButton,
    kParentAtomIsContainer, kQTEventMouseClicked, 0, NULL, 0, 0, NULL,
    kGoToBeginningButtonDownIndex, NULL);
AddSpriteSetImageIndexAction(myBeginActionButton,
    kParentAtomIsContainer, kQTEventMouseClickedEnd, 0, NULL, 0, 0,
    NULL, kGoToBeginningButtonUpIndex, NULL);
AddMovieGoToBeginningAction(myBeginActionButton,
    kParentAtomIsContainer, kQTEventMouseClickedEndTriggerButton);
AddSpriteSetVisibleAction(myBeginActionButton,
    kParentAtomIsContainer, kQTEventMouseEnter, 0, NULL, 0, 0, NULL,
    true, NULL);
AddSpriteSetVisibleAction(myBeginActionButton,
    kParentAtomIsContainer, kQTEventMouseExit, 0, NULL, 0, 0, NULL,
    false, NULL);
AddSpriteToSample(mySample, myBeginActionButton,
    kGoToBeginningSpriteID);
QTDisposeAtomContainer(myBeginActionButton);

// add go to prev button
myErr = QTCopyAtom(myPrevButton, kParentAtomIsContainer,
&myPrevActionButton);
if (myErr != noErr)
    goto bail;

AddSpriteSetImageIndexAction(myPrevActionButton,
    kParentAtomIsContainer, kQTEventMouseClicked, 0, NULL, 0, 0, NULL,
    kGoToPrevButtonDownIndex, NULL);
AddSpriteSetImageIndexAction(myPrevActionButton,
    kParentAtomIsContainer, kQTEventMouseClickedEnd, 0, NULL, 0, 0,
    NULL, kGoToPrevButtonUpIndex, NULL);
AddMovieStepBackwardAction(myPrevActionButton,
    kParentAtomIsContainer, kQTEventMouseClickedEndTriggerButton);
```

QTWiredSprite.c Sample Code

```

AddSpriteSetVisibleAction(myBeginActionButton,
    kParentAtomIsContainer, kQTEventMouseEnter, 0, NULL, 0, 0, NULL,
    true, NULL);
AddSpriteSetVisibleAction(myBeginActionButton,
    kParentAtomIsContainer, kQTEventMouseExit, 0, NULL, 0, 0, NULL,
    false, NULL);
AddSpriteToSample(mySample, myPrevActionButton, kGoToPrevSpriteID);
QTDisposeAtomContainer(myPrevActionButton);

// add go to next button
myErr = QTCopyAtom(myNextButton, kParentAtomIsContainer,
    &myNextActionButton);
if (myErr != noErr)
    goto bail;

AddSpriteSetImageIndexAction(myNextActionButton,
    kParentAtomIsContainer, kQTEventMouseClicked, 0, NULL, 0, 0, NULL,
    kGoToNextButtonDownIndex, NULL);
AddSpriteSetImageIndexAction(myNextActionButton,
    kParentAtomIsContainer, kQTEventMouseClickedEnd, 0, NULL, 0, 0,
    NULL, kGoToNextButtonUpIndex, NULL);
AddMovieStepForwardAction(myNextActionButton, kParentAtomIsContainer,
    kQTEventMouseClickedEndTriggerButton);
AddSpriteSetVisibleAction(myBeginActionButton,
    kParentAtomIsContainer, kQTEventMouseEnter, 0, NULL, 0, 0, NULL,
    true, NULL);
AddSpriteSetVisibleAction(myBeginActionButton,
    kParentAtomIsContainer, kQTEventMouseExit, 0, NULL, 0, 0, NULL,
    false, NULL);
AddSpriteToSample(mySample, myNextActionButton, kGoToNextSpriteID);
QTDisposeAtomContainer(myNextActionButton);

// add go to end button
myErr = QTCopyAtom(myEndButton, kParentAtomIsContainer,
    &myEndActionButton);
if (myErr != noErr)
    goto bail;

AddSpriteSetImageIndexAction(myEndActionButton,
    kParentAtomIsContainer, kQTEventMouseClicked, 0, NULL, 0, 0, NULL,
    kGoToEndButtonDownIndex, NULL);

```

QTWiredSprite.c Sample Code

```

AddSpriteSetImageIndexAction(myEndActionButton,
    kParentAtomIsContainer, kQTEventMouseClickedEnd, 0, NULL, 0, 0,
    NULL, kGoToEndButtonUpIndex, NULL);
AddMovieGoToEndAction(myEndActionButton, kParentAtomIsContainer,
    kQTEventMouseClickedEndTriggerButton);
AddSpriteSetVisibleAction(myBeginActionButton,
    kParentAtomIsContainer, kQTEventMouseEnter, 0, NULL, 0, 0, NULL,
    true, NULL);
AddSpriteSetVisibleAction(myBeginActionButton,
    kParentAtomIsContainer, kQTEventMouseExit, 0, NULL, 0, 0, NULL,
    false, NULL);
AddSpriteToSample(mySample, myEndActionButton, kGoToEndSpriteID);
QTDisposeAtomContainer(myEndActionButton);

// add penguin one
myErr = QTCopyAtom(myPenguinOne, kParentAtomIsContainer,
    &myPenguinOneAction);
if (myErr != noErr)
    goto bail;

// show the buttons on mouse enter and hide them on mouse exit
AddSpriteSetVisibleAction(myPenguinOneAction, kParentAtomIsContainer,
    kQTEventMouseEnter, 0, NULL, 0, kTargetSpriteID,
    (void *)kGoToBeginningSpriteID, true, NULL);
AddSpriteSetVisibleAction(myPenguinOneAction, kParentAtomIsContainer,
    kQTEventMouseExit, 0, NULL, 0, kTargetSpriteID,
    (void *)kGoToBeginningSpriteID, false, NULL);
AddSpriteSetVisibleAction(myPenguinOneAction, kParentAtomIsContainer,
    kQTEventMouseEnter, 0, NULL, 0, kTargetSpriteID,
    (void *)kGoToPrevSpriteID, true, NULL);
AddSpriteSetVisibleAction(myPenguinOneAction, kParentAtomIsContainer,
    kQTEventMouseExit, 0, NULL, 0, kTargetSpriteID,
    (void *)kGoToPrevSpriteID, false, NULL);
AddSpriteSetVisibleAction(myPenguinOneAction, kParentAtomIsContainer,
    kQTEventMouseEnter, 0, NULL, 0, kTargetSpriteID,
    (void *)kGoToNextSpriteID, true, NULL);
AddSpriteSetVisibleAction(myPenguinOneAction, kParentAtomIsContainer,
    kQTEventMouseExit, 0, NULL, 0, kTargetSpriteID,
    (void *)kGoToNextSpriteID, false, NULL);
AddSpriteSetVisibleAction(myPenguinOneAction, kParentAtomIsContainer,
    kQTEventMouseEnter, 0, NULL, 0, kTargetSpriteID,

```

APPENDIX C

QTWiredSprite.c Sample Code

```
(void *)kGoToEndSpriteID, true, NULL);
AddSpriteSetVisibleAction(myPenguinOneAction, kParentAtomIsContainer,
    kQTEventMouseExit, 0, NULL, 0, kTargetSpriteID,
    (void *)kGoToEndSpriteID, false, NULL);
AddSpriteToSample(mySample, myPenguinOneAction, kPenguinOneSpriteID);
QTDisposeAtomContainer(myPenguinOneAction);

// add penguin two
myErr = QTCopyAtom(myPenguinTwo, kParentAtomIsContainer,
    &myPenguinTwoAction);
if (myErr != noErr)
    goto bail;

// blink when clicked on
AddSpriteSetImageIndexAction(myPenguinTwoAction,
    kParentAtomIsContainer, kQTEventMouseClicked, 0, NULL, 0, 0, NULL,
    kPenguinClosedIndex, NULL);
AddSpriteSetImageIndexAction(myPenguinTwoAction,
    kParentAtomIsContainer, kQTEventMouseClickedEnd, 0, NULL, 0, 0,
    NULL, kPenguinForwardIndex, NULL);

AddQTEventAtom(myPenguinTwoAction, kParentAtomIsContainer,
    kQTEventMouseClickedEndTriggerButton, &myEventAtom);

// toggle the movie rate and both of the birds' graphics modes
QTWired_AddPenguinTwoConditionalActions(myPenguinTwoAction,
    myEventAtom);

QTWired_AddWraparoundMatrixOnIdle(myPenguinTwoAction);

AddSpriteToSample(mySample, myPenguinTwoAction, kPenguinTwoSpriteID);
QTDisposeAtomContainer(myPenguinTwoAction);

// add an action for when the key frame is loaded, to set the movie's
// looping mode to palindrome;
// note that this will actually be triggered every time the key frame
// is reloaded,
// so if the operation was expensive we could use a conditional to
// test if we've already done it
myLoopingFlags = loopTimeBase | palindromeLoopTimeBase;
AddMovieSetLoopingFlagsAction(mySample, kParentAtomIsContainer,
```

APPENDIX C

QTWiredSprite.c Sample Code

```
kQTEventFrameLoaded, myLoopingFlags);

// add the key frame sample to the sprite track media
//
// to add the sample data in a compressed form, you would use a
// QuickTime DataCodec to perform the
// compression; replace the call to the utility
// AddSpriteSampleToMedia with a call to the utility
// AddCompressedSpriteSampleToMedia to do this

AddSpriteSampleToMedia(myMedia, mySample, kSpriteMediaFrameDuration,
    true, NULL);
//AddCompressedSpriteSampleToMedia(myMedia, mySample,
// kSpriteMediaFrameDuration, true, zlibDataCompressorSubType, NULL);

//////////
//
// add a few override samples to move penguin one and change its
// image index
//
//////////

// original penguin one location
myLocation.h = (3 * kSpriteTrackWidth / 8) - (kPenguinWidth / 2);
myLocation.v = (kSpriteTrackHeight / 4) - (kPenguinHeight / 2);

myDelta = (kSpriteTrackHeight / 2) / kNumOverrideSamples;
myIndex = kPenguinDownRightCycleStartIndex;

for (i = 1; i <= kNumOverrideSamples; i++) {
    QTRemoveChildren(mySample, kParentAtomIsContainer);
    QTNewAtomContainer(&myPenguinOneOverride);

    myLocation.h += myDelta;
    myLocation.v += myDelta;
    myIndex++;
    if (myIndex > kPenguinDownRightCycleEndIndex)
        myIndex = kPenguinDownRightCycleStartIndex;

    SetSpriteData(myPenguinOneOverride, &myLocation, NULL, NULL,
        &myIndex, NULL, NULL, NULL);
```


APPENDIX C

QTWiredSprite.c Sample Code

```
        AddSpriteToSample(mySample, myPenguinOneOverride,
                           kPenguinOneSpriteID);
        AddSpriteSampleToMedia(myMedia, mySample,
                                kSpriteMediaFrameDuration, false, NULL);
        QTDisposeAtomContainer(myPenguinOneOverride);
    }

    EndMediaEdits(myMedia);

    // add the media to the track
    InsertMediaIntoTrack(myTrack, 0, 0, GetMediaDuration(myMedia),
                          fixed1);

    ////////////
    //
    // set the sprite track properties
    //
    ////////////
    {
        QTAtomContainer    myTrackProperties;
        RGBColor           myBackgroundColor;

        // add a background color to the sprite track
        myBackgroundColor.red = EndianU16_NtoB(0x8000);
        myBackgroundColor.green = EndianU16_NtoB(0);
        myBackgroundColor.blue = EndianU16_NtoB(0xffff);

        QTNewAtomContainer(&myTrackProperties);
        QTInsertChild(myTrackProperties, 0,
                      kSpriteTrackPropertyBackgroundColor, 1, 1,
                      sizeof(RGBColor), &myBackgroundColor, NULL);

        // tell the movie controller that this sprite track has actions
        hasActions = true;
        QTInsertChild(myTrackProperties, 0,
                      kSpriteTrackPropertyHasActions, 1, 1,
                      sizeof(hasActions), &hasActions, NULL);

        // tell the sprite track to generate QTIdleEvents
        myFrequency = EndianU32_NtoB(60);
        QTInsertChild(myTrackProperties, 0,
```

APPENDIX C

QTWiredSprite.c Sample Code

```
        kSpriteTrackPropertyQTIdleEventsFrequency, 1, 1,
        sizeof(myFrequency), &myFrequency, NULL);
myErr = SetMediaPropertyAtom(myMedia, myTrackProperties);
if (myErr != noErr)
    goto bail;

    QTDisposeAtomContainer(myTrackProperties);
}

//////////
//
// finish up
//
//////////

// add the movie resource to the movie file
myErr = AddMovieResource(myMovie, myResRefNum, 0,
                        myReply.sfFile.name);

bail:
    if (mySample != NULL)
        QTDisposeAtomContainer(mySample);

    if (myBeginButton != NULL)
        QTDisposeAtomContainer(myBeginButton);

    if (myPrevButton != NULL)
        QTDisposeAtomContainer(myPrevButton);

    if (myNextButton != NULL)
        QTDisposeAtomContainer(myNextButton);

    if (myEndButton != NULL)
        QTDisposeAtomContainer(myEndButton);

    if (myResRefNum != 0)
        CloseMovieFile(myResRefNum);

    if (myMovie != NULL)
        DisposeMovie(myMovie);
```

APPENDIX C

QTWiredSprite.c Sample Code

```
        return(myErr);
    }

    ////////////
    //
    // QTWired_AddPenguinTwoConditionalActions
    // Add actions to the second penguin that transform him (her?) into a two
    // state button
    // that plays or pauses the movie.
    //
    // We are relying on the fact that a "GetVariable" for a variable ID
    // which has never been set
    // will return zero. If we needed a different default value, we could
    // initialize it using the
    // frameLoaded event.
    //
    // A higher-level description of the logic is:
    //
    // On MouseUpInside
    //     If (GetVariable(DefaultTrack, 1) = 0)
    //         SetMovieRate(1)
    //         SetSpriteGraphicsMode(DefaultSprite, { blend, grey } )
    //         SetSpriteGraphicsMode(GetSpriteByID(DefaultTrack, 5),
    //             { ditherCopy, white } )
    //         SetVariable(DefaultTrack, 1, 1)
    //     ElseIf (GetVariable(DefaultTrack, 1) = 1)
    //         SetMovieRate(0)
    //         SetSpriteGraphicsMode(DefaultSprite, { ditherCopy, white })
    //         SetSpriteGraphicsMode(GetSpriteByID(DefaultTrack, 5),
    //             { blend, grey })
    //         SetVariable(DefaultTrack, 1, 0)
    //     Endif
    // End
    //
    ////////////

    OSErr QTWired_AddPenguinTwoConditionalActions (QTAtomContainer
                                                    theContainer, QTAtom theEventAtom)
    {
        QTAtom          myNewActionAtom, myNewParamAtom, myConditionalAtom;
```

APPENDIX C

QTWiredSprite.c Sample Code

```
QTAtom          myExpressionAtom, myOperatorAtom, myActionListAtom;
short           myParamIndex, myConditionIndex, myOperandIndex;
float           myConstantValue;
QTAtomID        myVariableID;
ModifierTrackGraphicsModeRecord    myBlendMode, myCopyMode;
OSErr           myErr = noErr;

myBlendMode.graphicsMode = blend;
myBlendMode.opColor.red = myBlendMode.opColor.green =
    myBlendMode.opColor.blue = 0x8fff;          // grey

myCopyMode.graphicsMode = ditherCopy;
myCopyMode.opColor.red = myCopyMode.opColor.green =
    myCopyMode.opColor.blue = 0xffff;          // white

AddActionAtom(theContainer, theEventAtom, kActionCode,
    &myNewActionAtom);

myParamIndex = 1;
AddActionParameterAtom(theContainer, myNewActionAtom, myParamIndex,
    0, NULL, &myNewParamAtom);

// first condition
myConditionIndex = 1;
AddConditionalAtom(theContainer, myNewParamAtom, myConditionIndex,
    &myConditionalAtom);
AddExpressionContainerAtomType(theContainer, myConditionalAtom,
    &myExpressionAtom);
AddOperatorAtom(theContainer, myExpressionAtom, kOperatorEqualTo,
    &myOperatorAtom);

myOperandIndex = 1;
myConstantValue = kButtonStateOne;
AddOperandAtom(theContainer, myOperatorAtom, kOperandConstant,
    myOperandIndex, NULL, myConstantValue);

myOperandIndex = 2;
myVariableID = kPenguinStateVariableID;
AddVariableOperandAtom(theContainer, myOperatorAtom, myOperandIndex,
    0, NULL, 0, myVariableID);
```

APPENDIX C

QTWiredSprite.c Sample Code

```
AddActionListAtom(theContainer, myConditionalAtom,
                  &myActionListAtom);
AddMovieSetRateAction(theContainer, myActionListAtom, 0,
                  Long2Fix(1));
AddSpriteSetGraphicsModeAction(theContainer, myActionListAtom, 0, 0,
                  NULL, 0, 0, NULL, &myBlendMode, NULL);
AddSpriteSetGraphicsModeAction(theContainer, myActionListAtom, 0, 0,
                  NULL, 0, kTargetSpriteID, (void *)kPenguinOneSpriteID,
                  &myCopyMode, NULL);
AddSpriteTrackSetVariableAction(theContainer, myActionListAtom, 0,
                  kPenguinStateVariableID, kButtonStateTwo, 0, NULL, 0);

// second condition
myConditionIndex = 2;
AddConditionalAtom(theContainer, myNewParamAtom, myConditionIndex,
                  &myConditionalAtom);
AddExpressionContainerAtomType(theContainer, myConditionalAtom,
                  &myExpressionAtom);
AddOperatorAtom(theContainer, myExpressionAtom, kOperatorEqualTo,
                  &myOperatorAtom);

myOperandIndex = 1;
myConstantValue = kButtonStateTwo;
AddOperandAtom(theContainer, myOperatorAtom, kOperandConstant,
                  myOperandIndex, NULL, myConstantValue);

myOperandIndex = 2;
myVariableID = kPenguinStateVariableID;
AddVariableOperandAtom(theContainer, myOperatorAtom, myOperandIndex,
                  0, NULL, 0, myVariableID);

AddActionListAtom(theContainer, myConditionalAtom,
                  &myActionListAtom);
AddMovieSetRateAction(theContainer, myActionListAtom, 0,
                  Long2Fix(0));
AddSpriteSetGraphicsModeAction(theContainer, myActionListAtom, 0, 0,
                  NULL, 0, 0, NULL, &myCopyMode, NULL);
AddSpriteSetGraphicsModeAction(theContainer, myActionListAtom, 0, 0,
                  NULL, 0, kTargetSpriteID, (void *)kPenguinOneSpriteID,
                  &myBlendMode, NULL);
```

APPENDIX C

QTWiredSprite.c Sample Code

```
        AddSpriteTrackSetVariableAction(theContainer, myActionListAtom, 0,
                                         kPenguinStateVariableID, kButtonStateOne, 0, NULL, 0);

bail:
    return(myErr);
}

//////////
//
// QTWired_AddWraparoundMatrixOnIdle
// Add beginning, end, and change matrices to the specified atom
// container.
//
//////////

OSErr QTWired_AddWraparoundMatrixOnIdle (QTAtomContainer theContainer)
{
    MatrixRecord    myMinMatrix, myMaxMatrix, myDeltaMatrix;
    long            myFlags = kActionFlagActionIsDelta |
                             kActionFlagParameterWrapsAround;
    QTAtom          myActionAtom;
    OSErr            myErr = noErr;

    myMinMatrix.matrix[0][0] = myMinMatrix.matrix[0][1] =
        myMinMatrix.matrix[0][2] = EndianS32_NtoB(0xffffffff);
    myMinMatrix.matrix[1][0] = myMinMatrix.matrix[1][1] =
        myMinMatrix.matrix[1][2] = EndianS32_NtoB(0xffffffff);
    myMinMatrix.matrix[2][0] = myMinMatrix.matrix[2][1] =
        myMinMatrix.matrix[2][2] = EndianS32_NtoB(0xffffffff);

    myMaxMatrix.matrix[0][0] = myMaxMatrix.matrix[0][1] =
        myMaxMatrix.matrix[0][2] = EndianS32_NtoB(0x7fffffff);
    myMaxMatrix.matrix[1][0] = myMaxMatrix.matrix[1][1] =
        myMaxMatrix.matrix[1][2] = EndianS32_NtoB(0x7fffffff);
    myMaxMatrix.matrix[2][0] = myMaxMatrix.matrix[2][1] =
        myMaxMatrix.matrix[2][2] = EndianS32_NtoB(0x7fffffff);

    myMinMatrix.matrix[2][1] = EndianS32_NtoB(Long2Fix((1 *
        kSpriteTrackHeight / 4) - (kPenguinHeight / 2)));
```

APPENDIX C

QTWiredSprite.c Sample Code

```
myMaxMatrix.matrix[2][1] = EndianS32_NtoB(Long2Fix((3 *
    kSpriteTrackHeight / 4) - (kPenguinHeight / 2)));

SetIdentityMatrix(&myDeltaMatrix);
myDeltaMatrix.matrix[2][1] = Long2Fix(1);

// change location
myErr = AddSpriteSetMatrixAction(theContainer,
    kParentAtomIsContainer, kQTEventIdle, 0, NULL, 0, 0, NULL,
    &myDeltaMatrix, &myActionAtom);
if (myErr != noErr)
    goto bail;

myErr = AddActionParameterOptions(theContainer, myActionAtom, 1,
    myFlags, sizeof(myMinMatrix), &myMinMatrix,
    sizeof(myMaxMatrix), &myMaxMatrix);

bail:
    return(myErr);
}
```

This Apple manual was written, edited, and composed on a desktop publishing system using Apple Mac OS computers and Adobe[™] FrameMaker software.

Line art was created using Adobe[™] Illustrator and Adobe Photoshop. PostScript[™], the page-description language for the LaserWriter, was developed by Adobe Systems Incorporated.

Text type is Palatino[®] and display type is Helvetica[®]. Bullets are ITC Zapf Dingbats[®]. Some elements, such as program listings, are set in Adobe Letter Gothic.

WRITER

Tom Maremaa

DEVELOPMENTAL EDITOR

Laurel Rezeau

ILLUSTRATOR

David Arrigoni

PRODUCTION EDITOR

Lorraine Findlay

SPECIAL THANKS TO

Sean Allen

Bruce Barrett

Deeje Cooley

Chris Flick

Steven Gulie

Michael Hinkson

Tim Monroe

George Towner

Bill Wright

ACKNOWLEDGMENTS TO

Jeff Mitchell