QuickTime Movie Playback Programming Guide

QuickTime > Movie Basics

2005-08-11

Contents

Figures and Listings

Introduction To QuickTime Movie Playback Programming Guide

This document describes how to open and play stored media, including QuickTime movies and other formats that QuickTime can automatically import and play, from sources such as a file, URL, pointer, or handle.

This document supersedes and replaces "Opening and Playing Movies."

The information in this document is of interest to nearly all developers working with QuickTime.

This document is intended for programmers working in C or C++ using the Carbon or QuickTime for Windows frameworks. Cocoa developers should generally use the QuickTime Kit framework instead, but may have occasion to use the interfaces described here for access to low-level functions not duplicated in Cocoa.

Organization of This Document

This document contains the following sections:

See Also

Before you read this document, you should already be familiar with *QuickTime Initialization Guide* and *QuickTime Overview*. If you are new to QuickTime, see Getting Started with QuickTime.

Overview

In QuickTime you play media of any kind—sound, video, still images, text, or animation—by creating a movie data structure in memory and playing the movie.

The movie is the fundamental data structure in QuickTime; it describes what media to present, where the media samples are located, and how and when to present them (duration, sequencing, compositing, layering, rotation and scaling, sound volume, and so on).

Important: Movies do not contain sample data, such as images or sound samples; movies contain **datareferences** that point to blocks of sample data. These data references can be file aliases, URLs, pointers, handles, or other types of references.

Playing a movie typically generates graphic output in a window, but other kinds of output are possible: a sound-only movie may not create a visible display; a movie may display using the whole screen instead of a window, or may render to an offscreen buffer or an external device such as a Firewire video recorder. Video output can be rendered directly to video memory as a series of OpenGL textures. Audio output can be directed to an external device or recorder.

In addition to playing existing QuickTime movies, QuickTime can automatically play media in over forty different formats and file types, including AIFF, WAVE, and MP3 audio; MPEG-1, MPEG-4, 3GPP, and DV video; JPEG, PNG, and GIF still images; and Flash and various other animation formats. For a full list, see "File Types that QuickTime Can Open as Movies" (page 43).

The movie or other media to be played can be stored in a file, a stream, or a buffer in memory, and can be accessed through a data reference such as a path and filename, file specification, URL, pointer, or handle.

Movies are typically stored in QuickTime movie files. The movie data samples are stored separately, either elsewhere in the same file or in other storage locations. When you get a movie from a QuickTime movie file, QuickTime copies the stored movie data structure into memory, then resolves the references to the sample data.

When you get a movie from a source other than a QuickTime movie file, such as a native MP3 or JPEG file, QuickTime creates a new movie describing the sample data; this is called **importinginplace,** because the sample data is not modified but remains in place. The import process consists of creating a movie data structure that describes the existing media.

Whether the movie is copied from storage or created on the fly by importing in place, it results in a data structure in memory that references data stored elsewhere. Once the data references are resolved, the movie can be played.

If the data must be downloaded from a remote storage location, the movie may go through the following load states:

loading—movie data structure not complete, or data references not resolved
playable—movie data structure complete; sample data downloading, but enough present to begin playing
playthrough OK—some sample data still downloading, but all data expected to arrive before it is needed
complete—all data available locally (any live streams are online)

To get and play a movie, take the following steps:

1. Choose a graphics destination, such as a visual context, a graphics world, or the default graphics port.
2. Optionally create an audio context if you want to select an audio output other than the default.
3. Specify a movie data source, such as a local file, a URL, or a block of memory referenced by a pointer or handle.
4. Get a movie from the source using one of several `NewMovieFrom...` functions, preferably `NewMovieFromProperties`. These functions create a movie in memory and return a valid movie reference. Use this reference with any function that takes a movie as a parameter.
5. Attach a movie controller component to the movie for playback.
6. Monitor the load state until the movie is playable, can play through without interruption, or is complete.
7. Grant the movie controller idle time. This can be automatic (using an `HIMovieView`), semiautomatic (using Carbon event timers), or manual (by granting idle time during your event loop).

Programmers working in version 10.4 and later of the Mac OS can use an `HIMovieView` object to control movie playback; this is the recommended technique.

Programmers working in QuickTime 6 for Mac OS X can use a Carbon movie controller. This is less desirable than using an `HIMovieView`.

Programmers working in the Windows OS or Mac OS 9 need to use the ActiveX control (Windows only) or a movie controller component. If you use an ActiveX control or a movie controller component, you need to implement a run loop that grants idle time to the control to enable movie playback.

Note: This document also describes low-level commands available for controlling movie playback directly, without using a movie controller, but this is *not* recommended practice.

Setting Up Graphics and Audio Output

Before you open a QuickTime movie, you need to decide where the movie output should go. If you want to use the default audio and video outputs, this is very simple (just verify that the default graphics output port is valid). Otherwise, you need to specify a graphics destination, an audio output, or both.

A graphics destination specifies where the visual output of a QuickTime movie is rendered. By default, QuickTime renders to the current thread's graphics port, which typically corresponds to a window, a view, or other control within a window. For simple playback, this default behavior is all you need; do not specify a graphics destination. Just verify that you have a valid graphics port and keep the graphics port valid for the life of the movie. For details, see "Using a Graphics Port or GWorld" (page 13).

QuickTime can also render to an offscreen buffer—either a `GWorld` buffer, a Core Video pixel buffer, or an OpenGL texture. Do this if you need to work with individual frames before, or instead of, presenting them to screen.

All versions of QuickTime can render to the default graphics device or a specified `GWorld`. QuickTime 7 and later can render to a Core Video pixel buffer or an OpenGL texture.

To work with individual video frames, you must specify a graphics destination by setting either a `GWorld` or a visual context. When you do this, you become responsible for final destination of the frames. You need to use a lower-level technology, such as QuickDraw, Core Image, or OpenGL, to transfer the images to the screen or other destination, and you are responsible for disposing of them when they are no longer needed.

To manipulate individual frames using a modern technology such as OpenGL, Core Video, or Core Image (Quartz), use a visual context to set the graphics destination by calling `NewMovieFromProperties` or `SetMovieVisualContext`. To manipulate video using older GWorld-based graphic systems such as QuickDraw, use `SetMovieGWorld`.

Mac OS X does not support direct rendering to screen. All rendering takes place to an offscreen buffer. If a `GWorld` or Core Video pixel buffer is specified, this is a buffer in main memory. If an OpenGL texture is specified, this is an image buffer in video memory. If no visual destination is specified, QuickTime attempts to use video memory to take advantage of graphics acceleration. The extent to which this is possible depends on such things as the media being rendered, the pixel formats, available transfer codecs, and the capabilities of the computer's graphics card.

For maximum performance, use the default graphics port or an OpenGL texture. Rendering to a `GWorld` or Core Video pixel buffer copies the image to and from main memory before final rendering, which slows things down.

Creating a Visual Context

Create a visual context if you need to manipulate individual frames before, or instead of, rendering them to the screen.

You create a visual context by calling one of the `QT...ContextCreate` functions, such as `QTPixelBufferContextCreate` or `QTOpenGLTextureContextCreate`. These functions return a `QTVisualContextRef`, an opaque token that you can pass to `NewMovieFromProperties` as part of a properties array or apply to an existing movie using `SetMovieVisualContext`.

Important: A visual context cannot be shared. You can free a visual context to be reused by another movie by setting the movie currently using the context to a null context.

`QTPixelBufferContextCreate` creates a visual context that causes QuickTime to render each frame to a Core Video pixel buffer in main memory. You can inspect or modify the frame and pass the pixel buffer to Core Image or OpenGL for display.

The `QTPixelBufferContextCreate` function takes a dictionary of attributes as an argument. This is a `CFDictionary`, an array of key-value pairs. This dictionary contains attributes such as the target dimensions and pixel buffer description. Some of these attributes are themselves contained in dictionaries, so the attributes dictionary can contain references to other dictionaries. Create the dictionaries using `CFDictionaryCreate`. For additional details, see "Visual Context Types" in *QuickTime 7 Update Guide*.

To specify a particular pixel buffer format, create a dictionary with the desired attributes as described in "Pixel Buffer Attribute Keys" in *Core Video Reference*. If you do not specify a pixel buffer format, QuickTime uses the native pixel format of the video frame.

To render each frame as an OpenGL texture, create the context using `QTOpenGLTextureContextCreate`. You are responsible for passing the textures to OpenGL for display.

`QTOpenGLTextureContextCreate` takes an OpenGL context and a pixel format object as arguments, as well as a `CFDictionary` of attributes.

To disable visual rendering, pass `NULL` instead of a `VisualContextRef`, either to `SetMovieVisualContext` or as a visual context movie property. This also frees the context for reuse or disposal.

To render to an offscreen graphics world instead of to a Core Video pixel buffer, disable visual rendering by setting a `NULL` visual context, then call `SetMovieGWorld` after the movie is instantiated. Do this if you need to manipulate individual frames using an older technology, such as QuickDraw, that works with GWorlds.

When you render to a pixel buffer or `GWorld`, you render to main memory and typically lose the advantages of graphics acceleration. When you render to an OpenGL texture, you render directly to video memory.

When you render to the default graphics port, QuickTime attempts to use video memory, but the actual rendering path depends on factors such as the media types, compressed pixel formats, and available transfer codecs.

To render to the default graphics port, do not include a visual context in the properties array when calling `NewMovieFromProperties`. You are not responsible for rendering individual frames in this case; QuickTime renders them automatically as the movie plays. For more information, see "Using a Graphics Port or GWorld" (page 13).

Note: As of QuickTime 7.0, the visual contexts available are Core Video pixel buffer, OpenGL texture, and `NULL`. Other visual contexts are likely to be added over time.

Using a Graphics Port or GWorld

If you instantiate a movie without specifying a visual context, the movie's graphics destination is set to your program's current graphics port (the port for the active window or thread). This is always the case when using versions of QuickTime prior to QuickTime 7 or when using `NewMovieFrom...` functions that do not accept a visual context.

You are not responsible for displaying individual frames when rendering to the default graphics port; this is handled automatically.

Important: If you use a graphics port instead of a visual context, your program's graphics port must be valid when the movie is created, even if the movie is sound-only. The graphics port must remain valid for the life of the movie (or until you set a different graphics destination for the movie using `SetMovieGWorld` or `SetMovieVisualContext`).

You can use `GetGWorld` to check for a valid port, and use `NewGWorld` to create a port if needed, before instantiating a movie.

A graphics port is automatically created when you create a window in the Mac OS. Your movie output typically goes to a view within a window, and therefore to the window's associated port; but if your application has not created a window, you may need to create a graphics port separately.

In the Windows OS, you create a valid graphics port using `NewGWorld` and associate it with your window by calling `CreatePortAssociation`. (You also need to call `DestroyPortAssociation` before disposing of your window.) Alternatively, you can create a `GWorld` that corresponds to your window's current graphics device by calling `GetNativeWindowPort`.

Changing a Movie's Graphics Destination

To change an existing movie's graphics destination to a new visual context, call `SetMovieVisualContext`.

To change an existing movie's graphics destination to a new `GWorld`, call `SetMovieGWorld.  If` the movie is currently using a visual context, free the context by calling `SetMovieVisualContext`, passing in `NULL` instead of a `VisualContextRef`, before setting the `GWorld`.

To disable visual rendering, call `SetMovieVisualContext`, passing in `NULL` instead of a `VisualContextRef`.

To get a movie's current graphics destination, call `GetMovieVisualContext`. If the movie's destination is a GWorld instead of a visual context, this function returns an error (`kQTVisualContextRequiredErr`); you can then call GetMovieGworld to obtain the `GWorld`.

`GetMovieGWorld` does *not* return an error for movies that use a visual context instead of a `GWorld`. It gives a special `GWorld` value that represents the visual context. This provides backward compatibility with existing code that gets a movie `GWorld` and later restores it.

Note: It is common practice to get a movie's current graphics destination, set the movie's graphics destination temporarily, then restore the original graphics destination. Prior to QuickTime 7, this was done by calling `GetMovieGWorld` to get a pointer to the current destination, then calling `SetMovieGWorld` to restore it. Existing code that uses these functions also works with movies that use a visual context.

Switching to Full-Screen Mode

When rendering to a graphics port, you can switch between playing your movie in a window and playing your movie using the full screen by calling `BeginFullScreen` and `EndFullScreen`. When playing a movie using the full screen, you should respond to the Escape key by ending full-screen mode.

Setting an Audio Context

QuickTime normally sends audio to the default audio output for your system. Channels are automatically mixed-down as needed (when playing multichannel sound through a stereo output, for example).

To direct the audio output of a movie to a particular device, or to set up a channel configuration or assign individual sound tracks to particular channels, create an audio context.

To create an audio context, call `QTAudioContextCreateForAudioDevice` and pass in the UID of an output device. An audio context reference is returned. Pass that audio context ref either to `NewMovieFromProperties`, as you would pass in a visual context, or to `SetMovieAudioContext`, to redirect the output of an existing movie.

Audio contexts are not shareable. If you want to route the output of two or more movies to the same device, call `QTAudioContextCreateForAudioDevice` once for each movie, passing in the same device UID to get another audio context for the same device. Pass a separate audio context reference to each movie.

You can pass `NULL` as the UID to `QTAudioContextCreateForAudioDevice` to create an audio context for the default audio output.

Note: As of this writing (QuickTime 7.01), audio contexts are not supported for MPEG audio tracks or streaming audio tracks.

Getting a Movie

To get a movie from a stored movie file or other data source, call one of the `NewMovieFrom...` functions, specifying the movie data source and any other new-movie properties, such as the graphics destination. The exact details of how you specify your graphics destination, movie data source, and other movie properties depends on which `NewMovieFrom...` function you call, which in turn depends on what version of QuickTime you are targeting.

- **QuickTime7** and later supports `NewMovieFromProperties`, which is the preferred `NewMovieFrom...` function. All propereties are passed as a movie properties array. The properties array is extensible and designed to accomodate future enhancements.
- **QuickTime6** and later supports `NewMovieFromDataRef`, which can also be used. Properties are set as a combination of parameters passed to `NewMovieFromDataRef` and properties set after the movie is instantiated. The list of properties that can be passed is not extensible. Audio and visual destinations, and several other properties, are always set to default values and can be changed only after the movie is created.
- **Olderversions** of QuickTime use a different `NewMovieFrom...` function for each type of movie data source (for example, `NewMovieFromFile`, `NewMovieFromHandle`, `NewMovieFromScrap`, and so on). These type-specific functions provide backward compatibility for existing code. Most are deprecated and should not be used when writing new applications, though a few remain useful for special purposes.

Getting a Movie in QuickTime 7 or Later

In QuickTime 7 and later, recommended practice is to get movies using the `NewMovieFromProperties` function. All the properties of the movie—graphics destination, movie data source, instantiation flags, and so on—are stored as elements in an array of movie properties. The properties array is then passed to `NewMovieFromProperties`.

The Movie Properties Array

The movie properties array is an extensible array that describes how the movie should be instantiated. Unlike a typical C structure (`struct`), new parameters can be added without breaking compatibility with existing code, and older versions of the properties array can be used with future versions of code. Any property not specified in the array is set to a default value. This means future versions of QuickTime will simply use default values for properites that can't be specified in code you write today. In addition, if a property is passed that QuickTime does not understand, or if the specified value cannot be set, an error is returned in the status field but the movie is still created. The calling application is free to determine whether this is a fatal error or an inconvenience. This will allow programs you write in the future to react appropriately if an earlier version of QuickTime does not recognize a new property that you set, regardless of what that new property is.

Each element in the movie properties array has a type field, identifying it as (for example) a movie data reference or a context. Each element also has a subtype ID field, identifying its data format, such as a `CFURL` or a QTVisualContextRef. These fields are followed by a data size field, indicating the number of bytes of data in the property value, and a pointer to the data. There is also a returned status field to indicate whether QuickTime was able to set the specified property value.

Field	Size
Type	4 bytes
ID	4 bytes
`DataSize`	4 bytes
`ValueAddr`	4 bytes
`PropStatus`	4 bytes

For example, you specify the graphics destination using a visual context,

In QuickTime 7 and later, you specify the graphics destination using a visual context, which has a movie property of type `kQTPropertyClass_Context`, and subtype ID of `kQTContextPropertyID_VisualContext`.

Similarly, you specify the movie data source as a movie property of type `kQTPropertyClass_DataLocation`. The subtype ID depends on the format of the data that specifies the movie source, such as a file specification, path, or URL.

`NewMovieFromProperties` also supports an audio context (kQTContextPropertyID_AudioContext). This allows you to specify an audio output device, set up a channel configuration, assign particular sound tracks to particular channels, and generally make use of the features of Core Audio, bypassing the limitations of the old Sound Manager. If no audio context is specified, the default output device and channel configuration are used.

Other settable movie properties include `kQTPropertyClass_MovieResourceLocator`, `kQTPropertyClass_MovieInstantiation`, and `kQTPropertyClass_NewMovieProperty`.

The `MovieResourceLocator` property lets you specify any special location of a stored movie data structure within a movie data source, such as a legacy Mac OS resource fork or an offset into a file containing multiple stored movies.

The `MovieInstantiation` property lets you control how QuickTime instantiates the movie. For example, you can tell QuickTime to ask the user for help if external data files can't be found, or tell QuickTime not to even look for external data files. You can tell `NewMovieFromProperties` to operate asynchronously, returning almost immediately and getting the movie in the background, or to operate synchronously, blocking until the movie is complete or an error has occured.

The `NewMovieProperty` property lets you specify additional movie characteristics, such as whether the movie is active or if it has a special default data reference for movie storage (for example, if the movie is not stored in a file, where QuickTime should initially offer to save the movie when the user chooses Save).

For additional details, see "Movie Property Constants" (page 51).

Specifying a Visual Context Property

Specify the graphics destination using a visual context, a movie property of type kQTPropertyClass_Context, and subtype ID of kQTContextPropertyID_VisualContext.

If this property is not specified, QuickTime renders movie output to the default graphics port for the thread. If you want QuickTime to render to the default graphics port, simply omit the visual context from the properties array.

Listing 1-1 shows how to create a visual context element and store it in a movie properties array.

Listing 3-1 Creating a visual context

```
// Allocate variables and pointers
CFAllocatorRef myAllocator;
CFDictionaryRef myAttributes;
QTVisualContextRef myContext;

// Create a key-value array pair with the pixel buffer attributes
// Create a CFDictionary from the arrays
//   Use default allocator and callbacks


// Create a CVPixelBuffer context
err = QTPixelBufferContextCreate(myAllocator, myAttributes, &myContext);

// Define a property array for NewMovieFromProperties
QTNewMoviePropertyElement movieProps[10];
ItemCount moviePropCount = 0;

// Add the visual context to the property array
movieProps[moviePropCount].propClass = kQTPropertyClass_Context;
movieProps[moviePropCount].propID = kQTContextPropertyID_VisualContext;
movieProps[moviePropCount].propValueSize = sizeof(myContext);
movieProps[moviePropCount].propValueAddress = &myContext;
movieProps[moviePropCount].propStatus = 0;
moviePropCount++;

// Add other movie properties to the array...
// ...
// Pass properties array to NewMovieFromProperties...
// ...
```

Specifying a Movie Data Source Property

The property class of a movie data source is `kQTPropertyClass_DataLocation`. The type of data reference (file specification, URL, and so forth) is passed in the property ID. See "Movie Property Constants" (page 51) for a complete list of data reference type IDs.

For example, a URL data source has the property ID `kQTDataLocationPropertyID_CFURL` (`'cfur'`), and its value is passed as a `CFURL`.

The code for passing a URL created from a literal string is shown in Listing 1-2.

Listing 3-2 Creating a URL from a string

```
// Define a property array for NewMovieFromProperties
QTNewMoviePropertyElement movieProps[10];
ItemCount moviePropCount = 0;

// Create a URL data reference of type CFURL
CFStringRefmyURLString = CFSTR("http://myserver.mydomain.com/myMovie.mov");
CFURLRef myURLRef = CFURLCreateWithString(kCFAllocatorDefault, myURLString,
Null);

// Add the movie data location to the property array
movieProps[moviePropCount].propClass = kQTPropertyClass_DataLocation;
movieProps[moviePropCount].propID = kQTDataLocationPropertyID_CFURL;
movieProps[moviePropCount].propValueSize = sizeof(myURLRef);
movieProps[moviePropCount].propValueAddress = (void*)&myURLRef;
movieProps[moviePropCount].propStatus = 0;
moviePropCount++;

// Add other movie properties to the array...
// ...
// Pass properties array to NewMovieFromProperties...
// ...
```

To specify a pointer or handle as the data source, you first need to create a data reference, as described in "Specifying a Movie Data Source Using a Data Reference" (page 21). You then pass the data reference to `NewMovieFromProperties` in the movie property array in the same way the `CFURLRef` is passed in Listing 3-2.

Getting a Movie Using NewMovieFromProperties

To use `NewMovieFromProperties`, you first create an array of movie properties set to your chosen values. These include the movie data source, the graphics destination, movie instantiation flags, whether the movie is active, and whether the movie supports user interaction. See "Movie Property Constants" (page 51) for a list of available movie properties.

You pass the array to `NewMovieFromProperties` and the movie is instantiated using the properties you specify. Unspecified properties are set to system defaults. To use the system default for a given property, simply omit that property from the array.

If no movie data source (`kQTPropertyClass_DataLocation`) is specified, the default is to create an empty movie.

If no audio or video context is specified (`kQTPropertyClass_Context`), the default is the current graphics port and audio output device.

If no resource locator is specified (`kQTPropertyClass_MovieResourceLocator`), the default is the first movie QuickTime finds in the source.

There are several movie instantiation flags implemented as properties of the class `kQTPropertyClass_MovieInstantiation`. The default behavior when these properties are omitted is as follows:

`kQTMovieInstantiationPropertyID_DontResolveDataRefs`—QuickTime attempts to resolve all data references.

kQTMovieInstantiationPropertyID_DontAskUnresolvedDataRefs—QuickTime asks the user to help if it cannot resolve references.

kQTMovieInstantiationPropertyID_DontAutoAlternates—Auto-alternates are automatically chosen (one alternate track is enabled; the other alternates are disabled).

kQTMovieInstantiationPropertyID_DontUpdateForeBackPointers

kQTMovieInstantiationPropertyID_AsyncOK—Loading is synchronous; NewMovieFromProperties blocks until either movie loading is complete or a fatal error is encountered.

kQTMovieInstantiationPropertyID_IdleImportOK—QuickTime may use a movie import component that operates asynchronously. See "Monitoring the Load State" (page 25)

kQTMovieInstantiationPropertyID_DontAutoUpdateClock

kQTMovieInstantiationPropertyID_ResultDataLocationChanged—QuickTime does not update the output property array if the data location of the movie changes.

If none of new-movie properties (kQTPropertyClass_NewMovieProperty) are specified, QuickTime uses its default new-movie settings: it copies the first movie it finds in the specified movie data source, the movie is not active when it opens, movie loading is synchronous, and the movie is set to interact with users.

While most of the QuickTime defaults are fine for normal movie playback, there are a few you will typically want to override:

You typically want to specify a data source, not create an empty movie.

You typically want to specify DontAskUnresolvedDataRef so QuickTime dones not ask the user for help locating files.

You typically want to pass the AsyncOK property so the function does not block during a long file download.

You typically want to pass the Active property so the movie is immediately active.

For additional details, see "Movie Property Constants" (page 51).

Listing 1-3 illustrates creating a movie properties array and passing it to NewMovieFromProperties.

Listing 3-3 Getting a movie using NewMovieFromProperties

```
// Define a property array for NewMovieFromProperties
QTNewMoviePropertyElement movieProps[10];
ItemCount moviePropCount = 0;

// Store the movie properties in the array

movieProps[moviePropCount].propClass = kQTPropertyClass_DataLocation;
movieProps[moviePropCount].propID = kQTDataLocationPropertyID_CFStringNativePath;
movieProps[moviePropCount].propValueSize = sizeof(inputPath);
movieProps[moviePropCount].propValueAddress = (void*)&inputPath;
movieProps[moviePropCount].propStatus = 0;
moviePropCount++;

movieProps[moviePropCount].propClass = kQTPropertyClass_MovieInstantiation;
```

```
movieProps[moviePropCount].propID =
kQTMovieInstantiationPropertyID_DontAskUnresolvedDataRefs;
movieProps[moviePropCount].propValueSize = sizeof(boolTrue);
movieProps[moviePropCount].propValueAddress = &boolTrue;
movieProps[moviePropCount].propStatus = 0;
moviePropCount++;

movieProps[moviePropCount].propClass = kQTPropertyClass_NewMovieProperty;
movieProps[moviePropCount].propID = kQTNewMoviePropertyID_Active;
movieProps[moviePropCount].propValueSize = sizeof(boolTrue);
movieProps[moviePropCount].propValueAddress = &boolTrue;
movieProps[moviePropCount].propStatus = 0;
moviePropCount++;

movieProps[moviePropCount].propClass = kQTPropertyClass_NewMovieProperty;
movieProps[moviePropCount].propID = kQTNewMoviePropertyID_DontInteractWithUser;
movieProps[moviePropCount].propValueSize = sizeof(boolTrue);
movieProps[moviePropCount].propValueAddress = &boolTrue;
movieProps[moviePropCount].propStatus = 0;
moviePropCount++;

movieProps[moviePropCount].propClass = kQTPropertyClass_Context;
movieProps[moviePropCount].propID = kQTContextPropertyID_VisualContext;
movieProps[moviePropCount].propValueSize = sizeof(visualContext);
movieProps[moviePropCount].propValueAddress = &visualContext;
movieProps[moviePropCount].propStatus = 0;
moviePropCount++;

movieProps[moviePropCount].propClass = kQTPropertyClass_Context;
movieProps[moviePropCount].propID = kQTContextPropertyID_AudioContext;
movieProps[moviePropCount].propValueSize = sizeof(audioContext);
movieProps[moviePropCount].propValueAddress = &audioContext;
movieProps[moviePropCount].propStatus = 0;
moviePropCount++;

if ((err = NewMovieFromProperties(moviePropCount, movieProps, 0, NULL, &movie)))
{
MyErrorHandlingRoutine;  // Your application's error handling routine
}

//  clean-up
DisposeMovie(movie); movie = NULL;
CFRelease(inputPath); inputPath = NULL;
QTVisualContextRelease(visualContext); visualContext = NULL;
CGLDestroyContext(cglContext); cglContext = NULL;
CGLDestroyPixelFormat(cglPixelFormat); cglPixelFormat = NULL;
QTAudioContextRelease(audioContext); audioContext = NULL;
```

To create an empty movie with no data source and default properties, pass in 0 and `NULL` as the movie properties count and movie properties array; for example:

```
// Create a blank movie using NewMovieFromProperties
movie myMovie;
err = NewMovieFromProperties(0, nil, 0, nil, &myMovie);
```

Getting a Movie in QuickTime 6

In QuickTime 6 neither the `NewMoviesFromProperties` function nor the audio and visual contexts are available. The graphics destination is the current graphics port, the audio output is the default output device, and recommended practice is to instantiate the movie using `NewMovieFromDataRef`.

Setting a Graphics Destination in QuickTime 6 and Earlier

In QuickTime 6 and earlier, a movie is always instantiated using the application's current graphics port, or `GWorld`. If you want to render to a different `GWorld`, you must change it by calling `SetMovieGWorld` after the movie is instantiated.

Your program's graphics port must be valid when the movie is created, even if the movie is sound-only. The graphics port must remain valid for the life of the movie (or until you set a different valid graphics port for the movie using `SetMovieGWorld`). To test for a valid graphics port, call `GetGWorld`.

A graphics port is automatically created and assigned when you create a window in the Mac OS.

In the Windows OS, you create a valid graphics port using `NewGWorld` and associate it with your window by calling `CreatePortAssociation`. (You also need to call `DestroyPortAssociation` before disposing your window.) You can create a `GWorld` that corresponds to your window's current graphics device in the Windows OS by calling `GetNativeWindowPort`.

Specifying a Movie Data Source Using a Data Reference

In QuickTime 6, you create a data reference for your movie data source (file, URL, pointer, or handle) and pass the data reference to the function `NewMovieFromDataRef`.

Note: Creating a new movie without a preexisting data source is a special case; use the function `NewMovie` to create an empty movie and `NewMovieFromUserProc` to create a movie whose data will be generated dynamically. These functions are discussed in depth in *QuickTime Movie Creation Guide*.

To create a data reference for the movie data source, use one of the `QTNewDataReferencer`s> functions:

- `QTNewDataReferenceFromFSRefCFString`
- `QTNewDataReferenceWithDirectoryCFString`
- `QTNewDataReferenceFromFullPathCFString`
- `QTNewDataReferenceFromURLCFString`
- `QTNewDataReferenceFromCFURL`
- `QTNewDataReferenceFromFSRef`
- `QTNewDataReferenceFromFSSpec`

There are functions you can use to create a data reference from a string representing a file system reference, a directory, a full path, or a URL. There are also functions for creating a data reference directly from a `CFURL`, `FSRef`, or `FSSpec`. Use the utility function that is most convenient for you. For example, if you have a C string that indicates a URL, convert the C string to a `CFString` and call `QTNewDataReferenceFromURLCFString`. Windows programmers with a native Windows pathname might find it more convenient to create a `CFString` and call `QTNewDataReferenceFromFullPathCFString`.

All of these functions take a pointer for a data reference and a pointer for a reference type. On return, these pointers indicate the location and type of data reference created; you pass these pointers to `NewMovieFromDataRef`.

Note: These `NewDataReference...` utility functions were added in QuickTime 6.4. You can create a data reference manually to create data references using earlier versions of QuickTime 6. For details, see the code sample "Example of Getting a Movie and Monitoring the Load State" (page 27) and Technical Note TN1195, Tagging Handle and Pointer Data References in QuickTime.

The data reference can point to either a QuickTime movie or to media data in a non-QuickTime format that QuickTime can import in place. If the data source does not contain a stored movie, QuickTime attempts to create a movie by searching the list of available movie import components based on the file type, MIME type, or filename extension of the data source.

If the data source is a buffer in memory, and is not a QuickTime movie, an additional step is required. Because your data reference is a pointer or handle, there is no MIME type, file type, or extension associated with it. In this case, you need to "tag" your data reference by adding one or more **datarefextensions,** such as a filename, file type, or MIME type, so QuickTime knows what type of importer is needed. For details, see Technical Note TN1195 Tagging Handle and Pointer Data References in QuickTime.

Getting a Movie Using NewMovieFromDataRef

`NewMovieFromDataRef` is a generalized routine that can instantiate a movie from most data sources. If you are writing an application that needs to run in QuickTime 6, it is recommended that you use `NewMovieFromDataRef` to instantiate the movie.

To use `NewMovieFromDataRef`, you first create a data reference to the movie data source. For details, see "Specifying a Movie Data Source Using a Data Reference" (page 21).

`NewMovieFromDataRef` also accepts a set of movie instantiation flags, indicating such things as whether the movie is active immediately, whether to query the user if QuickTime is unable to locate data files specified in the movie, and so on. For additional details, see `NewMovieFromDataRef`.

Listing 1-4 shows an example of creating a data reference and passing it to `NewMovieFromDataRef`, along with the movie instantiation flags.

Listing 3-4 Creating a data reference and calling NewMovieFromDataRef

```
// This function takes a CFStringRef to a full path and
// attempts to get a movie from the specified file
Booolean OpenMovie(CFStringRef inPath)
{
    Movie     myMovie = NULL;
    OSType    myDataRefType;
    Handle   myDataRef = NULL;
    short    myResID = 0;
    OSErr    myErr = noErr;

    ...

    // create the data reference
    myErr = QTNewDataReferenceFromFullPathCFString(inPath,
kQTNativeDefaultPathStyle,
                                                0, &myDataRef, &myDataRefType);
    if (myErr != noErr) goto bail;

    // get the Movie
    myErr = NewMovieFromDataRef(&myMovie,
newMovieActive | newMovieAsyncOK,
                                &myResID, myDataRef, myDataRefType);
    if (myErr != noErr) goto bail;

    // dispose of the data reference handle - we no longer need it
    DisposeHandle(myDataRef);

    // remember to call DisposeMovie when done with the returned Movie

    ...
}
```

Getting a Movie in Older Versions of QuickTime

For versions of QuickTime prior to QuickTime 6, you need to use a specific `NewMovieFrom...` function for each type of data source: `NewMovieFromFile` for files, `NewMovieFromHandle` for handles, `NewMovieFromScrap`, and so on.

Important: These functions are now deprecated and should not be used for new applications.

There is no general procedure for specifying a movie data source for these functions; every type of data source is a special case. Different functions require you to pass the data source in different ways. For example, to use `NewMovieFromFile`> you need to pass in an `FSRef`. This is commonly obtained by calling `OpenMovieFile` with an `FSSpec`. Windows users might first need to convert a native Windows pathname to a Mac OS `FSSpec` using the utility function `FSMakeFSSpec`, setting both the volume reference number and directory ID to 0 and passing the full pathname in place of a filename.

Details on the required format for the movie data source are provided in the documentation for the specific `NewMovie`rs> functions.

The graphics destination for movies created using these functions is always the current graphics port for your program. For additional details, see "Setting a Graphics Destination in QuickTime 6 and Earlier" (page 21).

The QuickTime documentation is rich in code samples that use these functions. For implementation details, see the *QuickTimeAPIReference* for these functions:

- `OpenMovieFile`

- `NewMovieFromFile`
- `CloseMovieFile`
- `NewMovieFromHandle`
- `NewMovieFromUserProc`
- `NewMovieFromDataFork`
- `NewMovieFromDataFork64`
- `NewMovieFromScrap`
- `NewMovieFromStorageOffset`

Monitoring the Load State

Calling any `NewMovieFrom...` function initiates a series of events that you can monitor by making calls to `GetMovieLoadState`. The initial load state is always `kMovieLoadStateLoading`, at least briefly. Eventually the load state changes to either `kMovieLoadStateComplete` or `kMovieLoadStateError`. There are often additional intermediate load states.

All `NewMovieFrom...` functions support asynchronous movie loading. This is controlled by passing a flag when calling the function. When asynchronous loading is enabled, the `NewMovieFrom...` function returns almost immediately, so that your application is not blocked while QuickTime locates the movie data source, resolves its media sample data references, downloads any necessary files, or performs any necessary import. You need to monitor the progress of the movie loading process by checking the load state periodically by calling `GetMovieLoadState`.

All `NewMovieFrom...` functions, including `NewMovieFromProperties`, load asynchronously only if the `newMovieAsyncOK` flag is passed (when calling `NewMovieFromProperties`, this flag is passed in the `NewMovieProperties` property).

If asynchronous loading is not used, any `NewMovieFrom...` function blocks until the movie finds and loads all referenced data or the process fails, and the load state resolves to either `kMovieLoadStateComplete` or `kMovieLoadStateError`.

The possible load states during asynchronous movie loading are these:

- `kMovieLoadStateLoading`—QuickTime still instantiating the movie.
- `kMovieLoadStatePlayable`—movie fully formed and can be played; media data still downloading.
- `kMovieLoadStatePlaythroughOK`—media data still downloading, but all data is expected to arrive before it is needed.
- `kMovieLoadStateComplete`—all media data is available.
- `kMovieLoadStateError`—movie loading failed; a movie may have been created, but it is not playable.

The Download Sequence

The exact sequence of events depends on whether the movie source is local or must be downloaded, whether it is self-contained or references external data, and whether it is a QuickTime movie or a source for which QuickTime uses an importer.

Local Movies

The simplest example of load states is a self-contained movie file on a local disk. QuickTime locates the file, finds and copies the movie data structure, then determines the location of the media sample data for each track. In this case, the data are all stored in the local movie file, so the movie is immediately complete and ready to play. The load state goes directly from `kMovieLoadStateLoading` to `kMovieLoadStateComplete`.

In the case of a movie file with external media data, QuickTime locates the file, finds and copies the movie data structure, then attempts to locate the necessary data files, streams, or storage containers.

Until all of the external data references are resolved, the load state remains `kMovieLoadStateLoading`. If all data references cannot be resolved, an error is returned, but the movie is still created in memory. It is not playable, however.

When QuickTime has resolved all the references to external data, the movie is ready to play. If all the data is in local files, the load state goes immediately to `kMovieLoadStateComplete`. If the data must download or stream over a network, the load state goes to `kMovieLoadStatePlayable`; at some point, depending on the network transmission speed and the movie's data rate, enough of the data downloads so that QuickTime expect to be able to play the movie without interruption (at the current transmission rate, the rest of the data will arrive before it is needed) and the load state changes to `kMovieLoadStatePlaythroughOK`.

When all the media samples referenced in the movie are available, the movie load state is `kMovieLoadStateComplete`. If the movie references data in a live stream, data may still be coming in, but all streams are online.

Movie Files on the Internet

A more complex example is a movie file downloaded over the Internet. QuickTime begins downloading the file and looking for the movie data structure (`kMovieLoadStateLoading`). Once the movie data structure has loaded, QuickTime can resolve any references to external media sample data. Once all data references are resolved, the movie state changes to `kMovieLoadStatePlayable`. As more media data arrives, the state changes to `kMovieLoadStatePlaythroughOK`, and finally to `kMovieLoadStateComplete` when all the data is available.

Note, however, that none of this takes place until the movie data structure has been downloaded. If the movie data structure is stored at the beginning of the file, as is normal, the sequence is as described. This is called **faststart,** or **progressivedownload,** because the movie is almost instantly playable and can be played through while it is still downloading. If the available bandwidth is greater than the movie bandwidth requirements, the movie state goes almost immediately to `kMovieLoadStatePlaythroughOK`.

But if the movie data structure is at the end of the file, the state remains `kMovieLoadStateLoading` until the entire file downloads. Not until then can QuickTime determine what types of media are used in the movie, where the movie sample data is located, or even if the file is actually a QuickTime movie file. If the movie file has no external data dependencies, the state goes suddenly from `kMovieLoadStateLoading` to `kMovieLoadStateComplete`.

Non-Movie Data Sources

In addition to QuickTime movie files, all of the `NewMovieFrom...` functions work with data sources that QuickTime can import in place, such as MP3 audio, MPEG-1 video, and so on. (For a complete list of formats that QuickTime can import in place, see "File Types that QuickTime Can Open as Movies" (page 43).)

If the data source does not contain a stored movie, QuickTime attempts to create a movie in memory that describes the sample data. This is typically done by searching the list of available movie import components, based on the file type, MIME type, or filename extension of the data source.

Not all movie import components support asynchronous loading; some components must have access to the entire file before they can begin providing movie data. Import components that support asynchronous loading are called **idlingimporters** (because the system grants them idle time after calling them). To enable movie import components to operate asynchronously, pass the `newMovieIdleImportOK` flag to the `NewMovieFrom...` function.

Once QuickTime has enough information to create an appropriate movie data structure for the data source, the movie state changes from `kMovieLoadStateLoading` to `kMovieLoadStateComplete` for a local file or a synchronous importer, or sequentially to `kMovieLoadStatePlayable`, `kMovieLoadStatePlaythroughOK`, then `kMovieLoadStateComplete` for a file that is downloading over the Internet through an idling importer.

If no importer can be found for the data source you specify, the movie load state changes to `kMovieLoadStateError`.

Example of Getting a Movie and Monitoring the Load State

Listing 1-5 illustrates getting a movie and monitoring the load state.

Listing 4-1 Getting a movie and monitoring the load state

```
Boolean CreateAMovie (void)
{
Point where = {100,100};
Movie theMovie = nil;
short resRefNum = 0;
short resId = movieInDataForkResID;
StringPtr fileName = QTUtils_ConvertCToPascalString ("MovieFile.mov");
Boolean isSelected = false;
Boolean isReplaceing = false;
OSErr err = noErr;
const unsigned char url[] =
"rtsp://a.movieserver.net/path/amovie.mov";  // create a dummy url for testing
HandledataRef = nil;
longloadState;
ComponentResult result;
Track firstProblemTrack;

// Create a data reference which we will use to instantiate our movie.
// In this case, we'll construct a URL data reference. The URL data reference
// is simply a handle whose data is a URL describing a movie.
// We'll build the data reference manually, just for the experience.

dataRef = NewHandleClear(StrLength(url) + 1);
CheckError(MemError(), "NewHandleClear error");
BlockMoveData(url, *dataRef, StrLength(url) + 1);

// Instantiate the movie file using NewMovieFromDataRef and a URL data reference.
// We make sure to pass the newMovieAsyncOK flag to enable
// us to query the state of the movie as it loads via the
// GetMovieLoadState function.

err = NewMovieFromDataRef(&theMovie,
newMovieActive | newMovieAsyncOK,
nil,
dataRef,
URLDataHandlerSubType);

// Handle asynchronous movie loading here. We use the
// GetMovieLoadState function to determine the load state
// of the movie.
do
{
long newLoadState;

// Get new load state to see if there was a change in
// state.
newLoadState = GetMovieLoadState(theMovie);
if (newLoadState != loadState)
{
loadState = newLoadState;
if (loadState < 0)
{
// failed to load the movie
// drop out of this loop and report an error
}

// movie loading--we need to keep tasking the movie so it gets
// time to load
MoviesTask(theMovie, 0);

if (loadState < kMovieLoadStatePlayable)
{
// movie still loading...
}

if (loadState < kMovieLoadStatePlaythroughOK)
{
// movie is playable but the media are still downloading...
}

if (loadState < kMovieLoadStateComplete)
{
// movie playable --
// some media still downloading, but all media is
// expected  to arrive before it is needed.
// we could start playing the movie now...
}
if (loadState >= kMovieLoadStateComplete)
{
// all media data is available
// drop out of this loop
// and play the movie
}
}
}
while ((loadState > kMovieLoadStateError) && (loadState <
kMovieLoadStateComplete));

CheckError(err, "NewMovieFromDataRef error");

// dispose of our data reference handle since it is no longer needed
DisposeHandle(dataRef);

result = GetMovieStatus (theMovie, &firstProblemTrack);
// if GetMovieLoadState from above returned kMovieLoadStateError, and
// GetMovieStatus returns nil for the firstProblemTrack parameter we
// know an error occurred
if ((loadState == kMovieLoadStateError) && (firstProblemTrack == nil))
{
CheckError(-1, "NewMovieFromDataRef error");
}

// create a default FSSpec
FSMakeFSSpec(0, 0L, fileName, &fsspec);

bail:
free(fileName);
return(err);
}
```

Playing a Movie

There are several different tool sets available for playing movies in QuickTime. The high-level tool sets all make use of a movie controller component; these components do most of the work required to play the movie, such as setting up any streams or buffers, resizing the movie, and redrawing the movie when it is moved or part of it is uncovered.

A movie controller component can include a human interface, allowing the user to start, stop, rewind, scrub, and adjust the sound volume, without you having to write any supporting code. This user interface need not be displayed, however.

You can override or augment any of the movie controller component functions with your own code by installing a callback function using `MCSetActionFilterWithRefCon`, and you can call any of the movie controller component's functions directly from your code (in response to input from your own custom user interface, for example). For details, see "Working Directly with Movie Controller Components" (page 39).

There are low-level commands in the Movie Toolbox that you can also use to control movies. This is usually neither necessary nor desirable, however, as it adds considerably to the complexity of your application. It also increases the risk that your application will not be able to take advantage of new features as they are added to QuickTime. If you use a movie controller component, the control code is updated with new versions of QuickTime and generally takes advantage of new features.

You can also enable a simple editing interface in a movie controller, giving the user the ability to perform cut-and-paste movie editing without requiring you to write supporting code. For more information, see *QuickTime Movie Basics*.

Once you have attached a movie controller component to your movie, to play the movie all you have to do is grant time to the controller.

In QuickTime 7 and later on Mac OS X, you can display the movie using an `HIMovieView`; the operating system automatically grants time to the controller as needed. You can create and initialize an `HIMovieView` using Interface Builder, greatly simplifying your code.

In QuickTime 6 and later on Mac OS X, you can display the movie using a Carbon movie controller. While somewhat more limited and more complex to set up than an `HIMovieView`, a Carbon movie controller also gets the processing time it needs automatically.

In older versions of QuickTime for Mac OS X, you can use a Carbon timer to generate an event whenever the movie needs processor time; you set a callback procedure that responds to this event by granting time to the controller and resetting the timer.

For versions of QuickTime running on Windows, or on Mac OS 9, you must periodically give idle time to the movie controller in your run loop.

Important: A movie controller component is not the same thing as the standard movie control bar. The standard movie control bar is one optional feature of movie controller components. You can use a movie controller component with any kind of interface, or no interface at all.

Note: Before you can play a movie, you need to either create a movie or get one from an external source. See "Getting a Movie" (page 15) and *QuickTime Movie Creation Guide*.

Playing a Movie Using HIMovieView

An `HIMovieView` is a specialized human interface object that displays QuickTime movies. It is a subclass of HIView, which is a subclass of HIObject. `HIMovieView` includes a movie controller component, a set of display coordinates, and more.

Note: HIObject is a common base class for all user interface objects. For Mac OS X version 10.2 and later, all menus, windows, controls, toolbars, and similar items, are subclasses of HIObject.

HIView is an object-oriented view system. Using the HIView model, every item within a window is a view: the root control, all embedded controls, the close, zoom, and minimize buttons, the resize control, and so on. Current Control Manager function calls are layered on top of this HIView model, so you can also control a view using existing Control Manager functions.

Some benefits of the HIView model include the following:

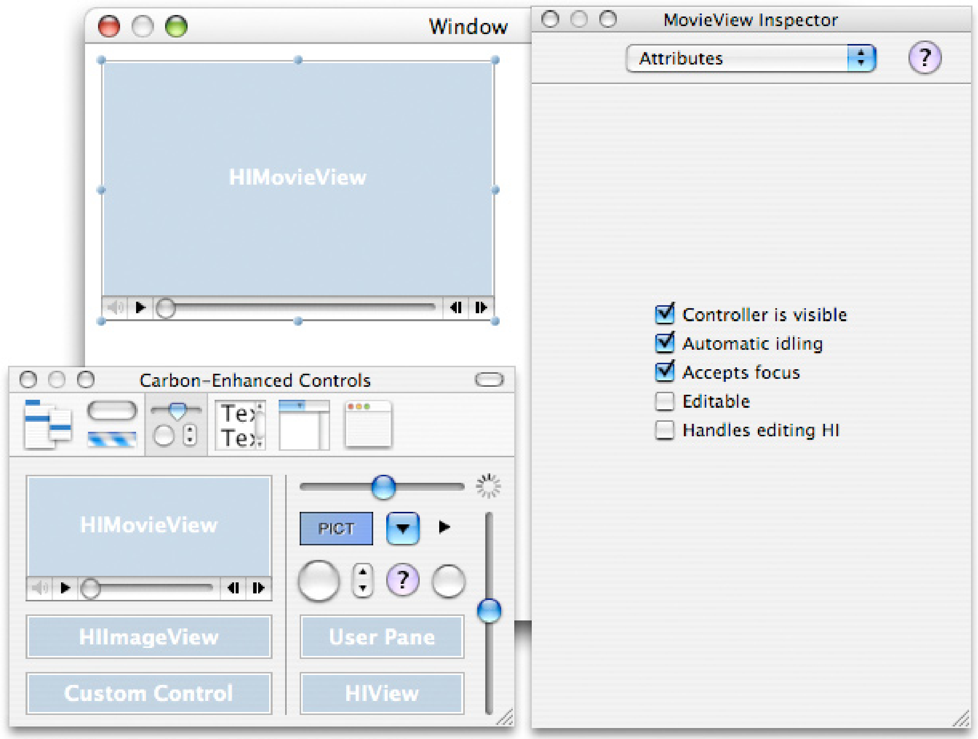
- Core Image (Quartz) is the native drawing system, but you can still use QuickDraw if desired.
- Coordinate system is not limited by the 16-bit space of QuickDraw. Floating point coordinates are valid.
- Simpified coordinate system for view bounds and view position within its parent.
- Views can be layered; you can easily place controls in front of, or behind, other controls.
- Views can be attached to, and detached from, parent windows, allowing you to move a view between windows.

Because `HIMovieView` is a subclass of HIView, it inherits these benefits over traditional controls.

Creating an HIMovieView

An `HIMovieView` can be created in Interface Builder by dragging an icon from a palette; you can then use the inspector window to set most of the viewing parameters without writing any code at all. In this case, you set the parent window, initial position of the movie within the window, and the initial size of the movie's display area, all in Interface Builder. There are attribute buttons to select whether the movie control bar is visible, whether the system automatically grants time to the movie as needed, whether the controller accepts keyboard input, whether the movie is editable, and whether the controller should enable its editing user interface as shown in Figure 1-1. All this information is stored in the nib file that specifies the parent window.

Figure 5-1 Creating an HIMovieView with Interface Builder



☑ Controller is visible
☑ Automatic idling
☑ Accepts focus
☐ Editable
☐ Handles editing HI

Alternatively, you can create and configure an `HIMovieView` entirely in code, in which case you must also set the various parameters, such as height and width, from your code when you instantiate the view. An example would look like this:

```
HIViewRef theHIMovieView;

// create an HIMovieview
HIObjectCreate(kHIMovieViewClassID, NULL, &theHIMovieView);

// set the movie attributes for the HIMovieView
HIMovieViewChangeAttributes(theHIMovieView, kHIMovieViewStandardAttributes, 0);

// attach my movie to the HIMovieView
HIMovieViewSetMovie(theHIMovieView, myMovie);
```

As a third alternative, you can create a generic HIView using Interface Builder, setting some of the view parameters there, then subclass it as an `HIMovieView` and configure it further from your code at runtime.

Important: A bug in `HIMovieViewCreate` makes it necessary to use this third method as a workaround in QuickTime version 7.0. See Technical Q&A QA1417, How to Work Around HIMovieViewCreate Failing for details. Later versions of QuickTime can use any of the three alternative methods.

If you create a generic HIView in Interface Builder (in this case with an ID of `'moov',0`, in a window named `MovieWindow`), your code would look something like this:

```
HIViewID  kMovieViewID = {'moov', 0};
```

```
HIViewRef theHIMovieView;
...

// get the window
CreateWindowFromNib(nib, CFSTR("MovieWindow"), &window);

// find our HIMovieView
HIViewFindByID(HIViewGetRoot(window), kMovieViewID, &theHIMovieView);

// set some attributes and a movie for the HIMovieView
HIMovieViewChangeAttributes(theHIMovieView, kHIMovieViewAutoIdlingAttribute |
                                        kHIMovieViewControllerVisibleAttribute,
                                             kHIMovieViewEditableAttribute |
                                           kHIMovieViewHandleEditingHIAttribute
 |
                                             kHIMovieViewAcceptsFocusAttribute);

HIMovieViewSetMovie(theHIMovieView, aMovie);

// install some event handlers
InstallWindowEventHandler(window, &MainWindowEventHandler,
                         GetEventTypeCount(windowEvents), windowEvents, window,
 NULL);

InstallHIObjectEventHandler((HIObjectRef)theHIMovieView, &HIMovieViewEventSniffer,
                             GetEventTypeCount(viewEvents), viewEvents, window,
 NULL);
// on with the show
ShowWindow(window);

...
```

Configuring an HIMovieView

An HIMovieView object responds to all the usual HIView commands, as well as Control Manager commands. For details, see HIView Programming Guide, and Control Manager Reference.

Use the standard HIView commands or the Control Manager commands to identify the view, find its parent window, change its visibility, and so on. For example, the GetControlBounds and SetControlBounds functions can be used to obtain and modify the view's size and location.

In addition to the standard HIView API, the following functions and constants are available specifically for use with HIMovieView:

Functions:

- HIMovieViewChangeAttributes—Changes the movie view attributes.
- HIMovieViewCreate—Creates an HIMovieView object.
- HIMovieViewGetAttributes—Returns the movie view's current attributes.
- HIMovieViewGetControllerBarSize—Returns the size of the visible movie control bar.
- HIMovieViewGetMovie—Returns the view's current movie.
- HIMovieViewGetMovieController—Returns the view's current movie controller.

- HIMovieViewPause—Pauses the view's current movie.
- HIMovieViewPlay—Plays the view's current movie.
- HIMovieViewSetMovie—Sets the view's current movie.

Constants:

```
kEventClassMovieView
kEventMovieViewOptimalBoundsChanged
kHIMovieViewClassID
kHIMovieViewAcceptsFocusAttribute
kHIMovieViewAutoIdlingAttribute
kHIMovieViewControllerVisibleAttribute
kHIMovieViewEditableAttribute
kHIMovieViewHandleEditingHIAttribute
kHIMovieViewNoAttributes
kHIMovieViewStandardAttributes
```

Use `HIMovieViewChangeAttributes` to set the HIMovieView's unique movie controller attributes, such as whether it allows editing or provides a user interface.

Use `HIMovieViewSetMovie` to attach a movie to the view.

Use `HIMovieViewPlay` and `HIMovieViewPause` to play and pause the movie programmatically.

Use `kHIMovieViewDataMovieController` to obtain the movie controller component instance if you need to communicate with the movie controller directly. You can then address commands to the movie controller component directly. For details, see "Working Directly with Movie Controller Components" (page 39).

You can also configure an `HIMovieView` object using Control Manager commands such as `SetControlData`. In addition to the applicable standard control parameters, an `HIMovieView` accepts the following constants with `SetControlData`:

```
kHIMovieViewDataMovieController       (read-only MovieController)
kHIMovieViewDataControllerBarHeight   (read-only UInt32)
kHIMovieViewDataMovie                 (Movie)
kHIMovieViewDataControllerVisible     (Boolean)
kHIMovieViewDataLocateTopLeft         (Boolean)
kHIMovieViewDataEditable              (Boolean)
kHIMovieViewDataHandleEditingHI       (Boolean)
kHIMovieViewDataKeysEnabled           (Boolean)
kHIMovieViewDataManualIdling          (Boolean)
```

To override or augment any of the functions `HIMovieView` performs, add an event handler using `InstallHIObjectEventHandler`. To override the controller, handle the associated event and mark it as handled; to augment the controller, perform the augmenting action but do not mark the event as handled. Alternatively, you can subclass the `HIMovieView` and create a custom view that inherits the features of `HIMovieView` as a base feature set.

HIMovieView and Your Run Loop

You don't actually need to do anything special in your run loop to play a movie in an `HIMovieView`. Just attach a movie using `HIMovieViewSetMovie`, set the auto-idling and visible-controller attributes (`kHIMovieViewAutoIdlingAttribute` and `kHIMovieViewControllerVisibleAttribute`), and `HIMovieView` does the rest.

Playing a Movie Using a Carbon Movie Control

QuickTime 6 for Mac OS X introduced the Carbon movie control. The `CreateMovieControl` function takes a movie and a window as arguments and creates a control containing a standard movie controller component. The movie control can then act as a Carbon event target, receiving Carbon events and dispatching them to its movie controller.

In other words, the Carbon movie control is implemented as a custom control that includes an event handler that handles any Carbon events sent to controls. When a Carbon movie control is created for a movie, a movie controller component instance is also created. The movie control then directs user interface events to its movie controller.

Your application can install additional event handlers on the Carbon movie control to handle such things as contextual menu clicks or to intercept events to do special processing. Control Manager calls can be made as well.

The movie controller is automatically idled by means of an event loop timer, using the Idle Manager to optimize idling frequency.

Creating and Configuring a Carbon Movie Control

Create a Carbon movie control by calling `CreateMovieControl`, passing in the parent window, position of the controller within the window, the movie to play, and the movie controller options, such as whether to make the control bar visible or enable editing.

The control's position in the window is passed as a `Rect` using local coordinates of the parent window. The movie control is centered within the specified rectangle by default and sized to fit within it while maintaining the movie's aspect ratio. If `NULL` is passed, the movie control is positioned at 0,0 in the window and has the natural dimensions of the movie (plus height of the movie control bar, if visible).

You can pass the following flags in the options field:

- `kMovieControlOptionHideController`—The movie controller is hidden when the movie control is drawn.
- `kMovieControlOptionLocateTopLeft`—The movie is pinned to the top left of `localRect` rather then being centered within it.
- `kMovieControlOptionEnableEditing`—Allows programmatic editing of the movie and enables drag and drop.
- `kMovieControlOptionHandleEditingHI`—Installs event handler for Edit menu commands and menu updating (also asserts `kMovieControlOptionEnableEditing`).

- `kMovieControlOptionSetKeysEnabled`—Allows the movie control to react to keystrokes and participate in the keyboard focus mechanism within the window.
- `kMovieControlOptionManuallyIdled`—Rather than being idled by the movie control event loop timer, this movie control is idled by the application, manually.

Once instantiated, a Carbon movie control accepts the standard Control Manager commands. For example, the `GetControlBounds` and `SetControlBounds` functions can be used to obtain and modify the movie control's size and location. See *Control Manager Reference* for more information.

In addition to the standard Control Manager selectors, you can use the following selectors for the `GetControlData` and `SetControlData` routines with Carbon movie controls:

- `kMovieControlDataMovieController`—(read only) Use with `GetControlData` to obtain the movie controller instance. You can then address the movie controller component directly using `MCDoAction`.

Important: Do not dispose of this movie controller; it is owned by the movie control it is associated with. You also must not use `MCSetControllerAttached` to detach the controller from the movie.

- `kMovieControlDataMovie`—(read only) Obtain the movie associated with a movie control after its creation.
- `kMovieControlDataManualIdling`—Obtain or modify the state of the movie control's idling behavior. By default, all movie controls are given time by the movie control event loop timer. If this Boolean item is `TRUE`, the application is required to give time to the movie controller in its run loop, using the `MCIdle` function.

Carbon Movie Controls and Your Run Loop

You do not need to do anything special in your run loop to play movies using a Carbon movie control, provided you enable the movie control's user interface and do not set the manual idle flag. The Carbon movie control takes care of all event routing to the movie. In order to distribute time to these movies, an event loop timer is set up which "idles" all movie controllers associated with Carbon movie controls within the application.

Playing a Movie Using a Movie Controller Component

If you are working in Windows OS or compiling for a version of QuickTime prior to QuickTime 6, you do not have access to either `HIMovieView` or Carbon movie controls. Consequently, you need to work with movie controller components directly. This is more complicated than playing a movie with an `HIMovieView` or a Carbon movie control, but it is still relatively straightforward; there's just more housekeeping.

You need to decide where your visual output should appear in the parent window and how much space to give it. This defines the movie controller's bounding box, the rectangle that encompasses the movie display area, and the movie control bar. You can get the movie's natural bounds and set the size of the display area to match, or choose an arbitrary display area and scale the movie to fit. If the movie is too large for the bounding box, it will be scaled down automatically, but it will only be scaled up to fill the display area if you request this behavior.

Once you've decided where your movie should appear, you open a movie controller component and attach it to your movie. You then provide time to the movie controller from within your application run loop.

Listing 1-5 is an example of opening a movie controller component and using it to play a movie.

Listing 5-1 Using a movie controller component to play a movie

```
  WindowPtr     myWindow = NULL;
  Rect          myBounds = {50, 50, 100, 100};
  Rect          myRect;
  StringPtr     myTitle = QTUtils_ConvertCToPascalString(kWindowTitle);
  OSErr         myErr;
  MovieController         gController;

// Initialize QuickTime
  myErr = EnterMovies();
  if (myErr != noErr)
goto bail;

// Create a display window
  myErr = memFullErr;
  myWindow = NewCWindow(NULL, &myBounds, myTitle, false, 0, (WindowPtr)-1, false,
 0);
  if (myWindow == NULL)
    goto bail;
  myErr = noErr;

// Align the movie bounding box with the display window
GetMovieBox(theMovie, &myRect); /* Get the movie box. */
OffsetRect(&myRect, -myRect.left, -myRecct.top); /* topleft=0 */
SetMovieBox(theMovie, &myRect);


// Get a movie controller component and attach it to the movie
// Pin the movie to the top left corner
 gController = NewMovieController (theMovie, &myBounds, mcTopLeftMovie);
        if (gController == nil)
goto bail;

// Get the movie controller's bounds and resize the window accordingly
 myErr = MCGetControllerBoundsRect (gController, &myRect);
 SizeWindow (myWindow, myRect.right, myRect.bottom, true);
 ShowWindow (myWindow);

// Enable the controller to accept keyboard focus
 myErr = MCDoAction (gController, mcActionSetKeysEnabled,(Ptr)true);

// Check for unforeseen errors
  myErr = GetMoviesError();
  if (myErr != noErr)
    goto bail;

// Application event loop goes here
// Idle the movie controller component inside your application event loop
// ...
  myErr = MCIdle(gController);
// ...
// On error or exit, dispose of the controller, movie, window (in that order)
bail:
  free(myTitle);

  if (gController != NULL)
DisposeMovieController(gController);

  if (theMovie != NULL)
    DisposeMovie(theMovie);

  if (myWindow != NULL)
    DisposeWindow(myWindow);
```

Note that you should not call `MCIsPlayerEvent` in Mac OS X, even though you may see this in some older sample code. Instead, use `MCIdle` or `MoviesTask` to grant time to the controller, or `MCKey` to have the movie controller handle a keystroke.

Working Directly with Movie Controller Components

Regardless of whether you play your movie using an `HIMovieView`, a Carbon movie control, or a naked movie controller component, your application can call the movie controller component instance directly and give it commands.

You obtain the component instance if you open the component directly using `NewMovieController`. If you use an HIMovieView, you can obtain the component instance by calling `HIMovieViewGetMovieController`. If you use a Carbon movie control, you can obtain the component instance by calling `GetControlData` with `kMovieControlDataMovieController`.

The following commands to the movie controller component are available to your application:

```
MCActivate
MCAdjustCursor
MCAddMovieSegment
MCClear
MCClick
MCCopy
MCCut
MCDoAction
MCDraw
MCDrawBadge
MCEnableEditing
MCGetClip
MCGetControllerBoundsRect
MCGetControllerBoundsRgn
MCGetControllerInfo
MCGetControllerPort
MCGetCurrentTime
MCGetDoActionsProc
MCGetIndMovie
MCGetInterfaceElement
MCGetMenuString
```

`MCGetVisible`
`MCGetWindowRgn`
`MCIdle`
`MCInvalidate`
`MCIsControllerAttached`
`MCIsEditingEnabled`
`MCIsPlayerEvent` (deprecated)
`MCKey`
`MCMovieChanged`
`MCNewAttachedController`
`MCPaste`
`MCPositionController`
`MCPtInController`
`MCRemoveAllMovies`
`MCRemoveAMovie`
`MCRemoveMovie`
`MCSetActionFilter`
`MCSetActionFilterWithRefCon`
`MCSetClip`
`MCSetControllerAttached`
`MCSetControllerBoundsRect`
`MCSetControllerCapabilities`
`MCSetControllerPort`
`MCSetDuration`
`MCSetIdleManager`
`MCSetMovie`
`MCSetUpEditMenu`
`MCSetVisible`
`MCTrimMovieSegment`
`MCUndo`

The `MCDoAction` function is of particular importance. `MCDoAction` can perform dozens of different operations, including starting and stopping the movie, setting the movie rate (playback speed), setting the audio volume, and so on. `MCDoAction` takes a selector as an argument. The selector tells `MCDoAction` which action to perform. For details, see `Movie Controller Actions`.

Obviously, all these functions allow you to do far more than just play a movie. They enable you to programmatically edit the movie, associate new movies with the controller, and configure the controller in various ways (enable editing, make the controller visible or invisible, and so on).

You can augment or override any controller function by installing a callback function, using `MCSetActionFilterWithRefCon`. See also “Using Movie Controllers” in the Programming Summary of *QuickTime API Reference*.

Important: The function `MCIsPlayerEvent` has been deprecated and should not be used on Mac OS X.

You will probably see references to `MCIsPlayerEvent` in sample code. In the past, `MCIsPlayerEvent` served two purposes: It granted processing time to the movie controller, and it allowed the movie controller to handle events such as mouse clicks and keystrokes. Both these features were based on the Apple Event Manager and event records, which are no longer used in Mac OS X.

If you use a Carbon movie control or an `HIMovieView`, processing time and event handling are taken care of automatically, so calls to `MCIsPlayerEvent` serve no purpose.

If your target system does not support Carbon movie controls or `HIMovieView`, you can call `MCIdle` to grant processing time to the movie controller, `MCClick` to have the movie controller handle a mouse click, and `MCKey` to have the movie controller handle a keystroke.

Bypassing the Movie Controller

It is possible to control a movie directly, without using an `HIMovieView`, a Carbon movie control, or a movie controller component. This is not recommended practice, however.

The low-level functions that allow you to work directly with movies can also be used when a movie controller of some sort is attached. This requires you to notify the controller that the movie has changed, and is also not recommended practice.

The low-level functions that work directly on movies are described in Appendix A, "Playing a Movie Using Low-Level Commands" (page 44).

Appendix A

This appendix contains a list of file types that QuickTime can open as movies and a list of functions that your application can use to control QuickTime movies without using a movie controller component.

File Types that QuickTime Can Open as Movies

QuickTime can open a number of different file types as movies. You can call a `NewMovieFrom...` function with a reference to one of these files and get back a playable movie. The list of file types is extensible by adding movie importer components (component type `'eat '`).

QuickTime can of course open QuickTime movie files (file extension `.mov`, MIME type `video/quicktime`) and QuickTime media link files (file extension `.qtl` or `.mov`, MIME type `application/x-quicktimeplayer` or `video/quicktime`). For more about QuickTime media link files, see *HTML Scripting Guide for QuickTime*.

QuickTime can also open any file type for which it has an installed importer. QuickTime ships with importers for dozens of file types, including:

3GPP
3GPP2
AIFF
AMC
AMR
Animated GIF
AU
AVI
BMP
DLS
DV
FlashPix
FLC
GIF
GSM
JPEG 2000 (Mac OS X)
JPEG/JFIF
Karaoke
MacPaint
Macromedia Flash 5
MIDI
MPEG-1
MP3 (MPEG-1 Layer 3)
M3U (MP3 Playlist files)
MPEG-4
M4A
M4B

M4P (iTunes 4 audio)
PDF (Mac OS X)
Photoshop
PICS
PICT
PLS
PNG
QCP
QuickTime Image File
SD2 (SoundDesigner 2)
SDP
SDV
SGI
SMIL
System 7 Sound (Mac OS 9)
Targa
Text
TIFF
TIFF Fax
VDU (Sony Video Disk Unit)
Wave

Additional import components, such as MPEG-2, can be purchased separately from Apple or are available from third parties.

Note that some file formats, such as MPEG-1, include a compression format. Other formats, such as AIFF, WAVE, and AVI, may contain data compressed using various schemes. QuickTime can open files that support multiple compression formats, but the file may not be playable unless QuickTime also has a compatible decompressor for the media. For example, an `.avi` file with MPEG-1 compressed video might open and play in QuickTime, while another `.avi` file compressed using WM9 compression would not play unless the user installed a WM9 decompressor component for QuickTime.

Playing a Movie Using Low-Level Commands

Just as the movie controller API sits below the Control Manager and `HIMovieView` API, a still lower-level API lets you control the movie directly. You can use these low-level Movie Toolbox calls in addition to, or instead of, a movie controller component.

As a rule, you should not play movies without a movie controller. Most of the operations that act directly on movies can also be performed using a movie controller by calling the `MCDoAction` function with the appropriate action selector. Do not play movies directly unless you have a compelling reason to do so; use a movie controller instead, or better yet, use a higher-level object that includes a movie controller, such as an `HIMovieView`.

Important: If you use commands that act directly on a movie that is attached to a controller, be sure to call `MCMovieChanged` before your next call to `MCIdle`. This lets the controller know that its internal record of the movie state needs to be updated.

If you play a movie without a movie controller component, your application is responsible for updating the movie, handling all user input, and handling events such as resizing or covering and uncovering the window. Your application starts the movie playing by calling `StartMovie`, then runs a loop until `IsMovieDone` returns `TRUE`. Inside the loop, your application calls `MoviesTask` and handles any resize, drag, cover/uncover, and UI events. An example of this kind of loop is shown in Listing 1-6.

Listing 6-1 Movie playing loop

```
#define doTheRightThing 5000

void PlayMyMovie (movie, aMovie)
{
    WindowPtr       aWindow;
    Rect            windowRect;
    Rect            movieBox;
    Movie           aMovie;
    Boolean         done = false;
    OSErr           err;
    EventRecord theEvent;
    WindowPtr       whichWindow;
    short           part;

    err = EnterMovies ();
    if (err) return;

    SetRect (&windowRect, 100, 100, 200, 200);
    aWindow = NewCWindow (nil, &windowRect, "\pMovie",
                                false, noGrowDocProc, (WindowPtr)-1,
                                true, 0);

    GetMovieBox (aMovie, &movieBox);
    OffsetRect (&movieBox, -movieBox.left, -movieBox.top);
    SetMovieBox (aMovie, &movieBox);

    SizeWindow (aWindow, movieBox.right, movieBox.bottom, true);
    ShowWindow (aWindow);
    SetMovieGWorld (aMovie, (CGrafPtr)aWindow, nil);

    StartMovie (aMovie);

    while ( !IsMovieDone(aMovie)  )
    {
// Handle resize and update events...
// Handle drag events

        MoviesTask (aMovie, DoTheRightThing);
    }
    DisposeMovie (aMovie);
    DisposeWindow (aWindow);
}
```

The Movie Toolbox commands for operating directly on movies are listed below. See the documentation on individual functions in *QuickTimeAPIReference* for more information.

```
StartMovie
StopMovie
GoToBeginningOfMovie
GoToEndOfMovie
MoviesTask
IsMovieDone
UpdateMovie
PtInMovie
PtInTrack
GetMovieStatus
GetTrackStatus
SetMovieActive
GetMovieActive
SetMovieGWorld
GetMovieGWorld
SetMovieBox
GetMovieBox
GetMovieDisplayBoundsRgn
GetMovieSegmentDisplayBoundsRgn
SetMovieDisplayClipRgn
GetMovieDisplayClipRgn
GetTrackDisplayBoundsRgn
GetTrackSegmentDisplayBoundsRgn
GetMovieDuration
SetMovieTimeValue
SetMovieTime
GetMovieTime
SetMovieRate
GetMovieRate
```

Movie Playback Reference

This section lists the functions, constants, and data types available for getting and playing QuickTime movies.

Error Functions

```
ClearMoviesStickyError
GetMoviesError
GetMoviesStickyError
SetMoviesErrorProc
```

FullScreen Functions

```
BeginFullScreen
EndFullScreen
```

See also: "Full Screen Flags" (page 53).

Event Loop Functions

```
DisposeQTNextTaskNeededSoonerCallbackUPP
GetMovieCoverProcs
GetMovieStatus
InvalidateMovieRegion
IsMovieDone
MoviesTask
NewQTNextTaskNeededSoonerCallbackUPP
PtInMovie
QTGetTimeUntilNextTask
QTInstallNextTaskNeededSoonerCallback
QTNextTaskNeededSoonerCallbackProc
QTUninstallNextTaskNeededSoonerCallback
SetMovieCoverProcs
SetMovieDrawingCompleteProc
UpdateMovie
```

Graphics Destination Functions

`CreatePortAssociation`
`GetMovieGWorld`
`GetMovieVisualContext`
GetPort
QTOpenGLTextureAvailableCallback
`QTOpenGLTextureContextCreate`
`QTPixelBufferContextCreate`
`QTVisualContextCopyImageForTime`
`QTVisualContextGetAttribute`
`QTVisualContextGetTypeID`
`QTVisualContextIsNewImageAvailable`
`QTVisualContextRelease`
`QTVisualContextRetain`
`QTVisualContextSetAttribute`
`QTVisualContextSetImageAvailableCallback`
`QTVisualContextTask`
`SetMovieGWorld`
`SetMovieVisualContext`
SetPort

Visual Playback Functions

`BeginFullScreen`
`EndFullScreen`
GetMovieVisualBrightness
GetMovieVisualContrast
GetMovieVisualHue
SetMovieVisualBrightness
SetMovieVisualContrast
SetMovieVisualHue
SetMovieVisualSaturation

Audio Playback Functions

`GetMovieAudioBalance`
GetMovieAudioContext
`GetMovieAudioFrequencyLevels`
`GetMovieAudioFrequencyMeteringBandFrequencies`
`GetMovieAudioFrequencyMeteringNumBands`
`GetMovieAudioGain`
`GetMovieAudioMute`
`GetMovieAudioVolumeLevels`
`GetMovieAudioVolumeMeteringEnabled`
`GetMovieVolume`

QTAudioContextCreateForAudioDevice
`QTAudioFrequencyLevels`
`QTAudioVolumeLevels`
`SetMovieAudioBalance`
`SetMovieAudioContext`
`SetMovieAudioFrequencyMeteringNumBands`
`SetMovieAudioGain`
`SetMovieAudioMute`
`SetMovieAudioVolumeMeteringEnabled`
`SetMovieVolume`

Functions that Get Movies from Storage

`NewMovieForDataRefFromHandle`
`NewMovieFromDataFork`
`NewMovieFromDataRef`
`NewMovieFromFile`
`NewMovieFromHandle`
`NewMovieFromProperties`
`NewMovieFromScrap`
`NewMovieFromStorageOffset`
`GetMovieLoadState`

HIMovieView Functions

`HIMovieViewChangeAttributes`
`HIMovieViewCreate`
HIMovieViewGetAttributes
HIMovieViewGetControllerBarSize
HIMovieViewGetMovie
`HIMovieViewGetMovieController`
`HIMovieViewPause`
`HIMovieViewPlay`
`HIMovieViewSetMovie`

Carbon Movie Controller Functions

See *Control Manager Reference*.

Additional selectors for Carbon Movie Controller:

`kMovieControlOptionHideController`
`kMovieControlOptionLocateTopLeft`
`kMovieControlOptionEnableEditing`
`kMovieControlOptionHandleEditingHI`

`kMovieControlOptionSetKeysEnabled`
`kMovieControlOptionManuallyIdled`
`kMovieControlDataMovieController`
`kMovieControlDataMovie`
`kMovieControlDataManualIdling`

Movie Controller Component Functions

`MCActivate`
`MCAdjustCursor`
`MCAddMovieSegment`
`MCClear`
`MCClick`
`MCCopy`
`MCCut`
`MCDoAction`
`MCDraw`
`MCDrawBadge`
`MCEnableEditing`
`MCGetClip`
`MCGetControllerBoundsRect`
`MCGetControllerBoundsRgn`
`MCGetControllerInfo`
`MCGetControllerPort`
`MCGetCurrentTime`
`MCGetDoActionsProc`
`MCGetIndMovie`
`MCGetInterfaceElement`
`MCGetMenuString`
`MCGetVisible`
`MCGetWindowRgn`
`MCIdle`
`MCInvalidate`
`MCIsControllerAttached`
`MCIsEditingEnabled`
`MCIsPlayerEvent` (deprecated)
`MCKey`
`MCMovieChanged`
`MCNewAttachedController`
`MCPaste`
`MCPositionController`
`MCPtInController`
`MCRemoveAllMovies`
`MCRemoveAMovie`
`MCRemoveMovie`
`MCSetActionFilter`
`MCSetActionFilterWithRefCon`
`MCSetClip`
`MCSetControllerAttached`
`MCSetControllerBoundsRect`

`MCSetControllerCapabilities`
`MCSetControllerPort`
`MCSetDuration`
`MCSetIdleManager`
`MCSetMovie`
`MCSetUpEditMenu`
`MCSetVisible`
`MCTrimMovieSegment`
`MCUndo`

Low-Level Movie Toolbox Functions

`AbortPrePrerollMovie`
`GetMovieActive`
`GetMovieActiveSegment`
`GetMovieBox`
`GetMovieCreationTime`
`GetMovieDisplayBoundsRgn`
`GetMovieModificationTime`
`GetMovieRate`
`GetMovieTime`
`GetMovieTimeBase`
`GetMovieTimeScale`
`GoToBeginningOfMovie`
`GoToEndOfMovie`
`LoadMovieIntoRam`
`PrePrerollMovie`
`PrerollMovie`
`SetMovieActive`
`SetMovieActiveSegment`
`SetMovieBox`
`SetMoviePlayHints`
`SetMovieRate`
`SetMovieTime`
`SetMovieTimeValue`
`StartMovie`
`StopMovie`

Constants and Selectors

The following constants and selectors are used when playing movies or using movie controllers.

Movie Property Constants

`kQTAudioMeter_DeviceMix`
`kQTAudioMeter_MonoMix`

kQTAudioMeter_StereoMix
kQTAudioPropertyID_Balance
kQTAudioPropertyID_BitRateString
kQTAudioPropertyID_ChannelLayoutString
kQTAudioPropertyID_DeviceChannelLayout
kQTAudioPropertyID_Fade
kQTAudioPropertyID_FormatString
kQTAudioPropertyID_Gain
kQTAudioPropertyID_Mute
kQTAudioPropertyID_SampleRateString
kQTAudioPropertyID_SampleSizeString
kQTAudioPropertyID_SummaryChannelLayout
kQTAudioPropertyID_SummaryString
kQTContextPropertyID_AudioContext
kQTContextPropertyID_VisualContext
kQTDataLocationPropertyID_CFStringHFSPath
kQTDataLocationPropertyID_CFStringNativePath
kQTDataLocationPropertyID_CFStringPosixPath
kQTDataLocationPropertyID_CFStringWindowsPath
kQTDataLocationPropertyID_CFURL
kQTDataLocationPropertyID_DataFork
kQTDataLocationPropertyID_DataReference
kQTDataLocationPropertyID_LegacyMovieResourceHandle
kQTDataLocationPropertyID_MovieUserProc
kQTDataLocationPropertyID_QTDataHandler
kQTDataLocationPropertyID_ResourceFork
kQTDataLocationPropertyID_Scrap
kQTMovieInstantiationPropertyID_AsyncOK
kQTMovieInstantiationPropertyID_DontAskUnresolvedDataRefs
kQTMovieInstantiationPropertyID_DontAutoAlternate
kQTMovieInstantiationPropertyID_DontAutoUpdateClock
kQTMovieInstantiationPropertyID_DontResolveDataRefs
kQTMovieInstantiationPropertyID_DontUpdateForeBackPointers
kQTMovieInstantiationPropertyID_IdleImportOK
kQTMovieInstantiationPropertyID_ResultDataLocationChanged
kQTMovieResourceLocatorPropertyID_Callback
kQTMovieResourceLocatorPropertyID_FileOffset
kQTMovieResourceLocatorPropertyID_LegacyResID
kQTMovieResourceLocatorPropertyID_LegacyResName
kQTNewMoviePropertyID_Active
kQTNewMoviePropertyID_DefaultDataRef
kQTNewMoviePropertyID_DontInteractWithUser
kQTPropertyClass_Context
kQTPropertyClass_DataLocation
kQTPropertyClass_MovieInstantiation
kQTPropertyClass_MovieResourceLocator
kQTPropertyClass_NewMovieProperty
kQTVisualContextColorSpaceKey
kQTVisualContextExpectedReadAheadKey
kQTVisualContextPixelBufferAttributesKey
kQTVisualContextTargetDimensionsKey
kQTVisualContextTargetDimensions_HeightKey
kQTVisualContextTargetDimensions_WidthKey

```
kQTVisualContextTypeKey
kQTVisualContextType_OpenGLTexture
kQTVisualContextType_PixelBuffer
```

Full Screen Flags

```
fullScreenAllowEvents
fullScreenCaptureAllDisplays
fullScreenCaptureDisplay
fullScreenDontChangeMenuBar
fullScreenDontSwitchMonitorResolution
fullScreenHideCursor
fullScreenPreflightSize
```

Movie Controller Component Constants and Selectors

```
kMCActivateSelect
kMCClearSelect
kMCClickSelect
kMCCopySelect
kMCCutSelect
kMCDoActionSelect
kMCDrawBadgeSelect
kMCDrawSelect
kMCEnableEditingSelect
kMCGetClipSelect
kMCGetControllerBoundsRectSelect
kMCGetControllerBoundsRgnSelect
kMCGetControllerInfoSelect
kMCGetControllerPortSelect
kMCGetCurrentTimeSelect
kMCGetMenuStringSelect
kMCGetVisibleSelect
kMCGetWindowRgnSelect
kMCIdleSelect
kMCIsControllerAttachedSelect
kMCIsEditingEnabledSelect
kMCIsPlayerEventSelect
kMCKeySelect
kMCMovieChangedSelect
kMCNewAttachedControllerSelect
kMCPasteSelect
kMCPositionControllerSelect
kMCRemoveMovieSelect
kMCSetActionFilterSelect
kMCSetActionFilterWithRefConSelect
kMCSetClipSelect
kMCSetControllerAttachedSelect
kMCSetControllerBoundsRectSelect
```

```
kMCSetControllerPortSelect
kMCSetDurationSelect
kMCSetMovieSelect
kMCSetUpEditMenuSelect
kMCSetVisibleSelect
kMCUndoSelect
mcActionActivate
mcActionBadgeClick
mcActionControllerSizeChanged
mcActionDeactivate
mcActionDraw
mcActionGetFlags
mcActionGetKeysEnabled
mcActionGetLoopIsPalindrome
mcActionGetLooping
mcActionGetPlayEveryFrame
mcActionGetPlayRate
mcActionGetPlaySelection
mcActionGetUseBadge
mcActionGetVolume
mcActionGoToTime
mcActionIdle
mcActionKey
mcActionMouseDown
mcActionMovieClick
mcActionPlay
mcActionResume
mcActionSetFlags
mcActionSetGrowBoxBounds
mcActionSetKeysEnabled
mcActionSetLoopIsPalindrome
mcActionSetLooping
mcActionSetPlayEveryFrame
mcActionSetPlaySelection
mcActionSetSelectionBegin
mcActionSetSelectionDuration
mcActionSetUseBadge
mcActionSetVolume
mcActionShowBalloon
mcActionStep
mcActionSuspend
mcFlagSuppressMovieFrame
mcFlagSuppressSpeakerButton
mcFlagSuppressStepButtons
mcFlagsUseWindowPalette
mcInfoClearAvailable
mcInfoCopyAvailable
mcInfoCutAvailable
mcInfoEditingEnabled
mcInfoHasSound
mcInfoIsInPalindrome
mcInfoIsLooping
mcInfoIsPlaying
```

```
mcInfoPasteAvailable
mcInfoUndoAvailable
mcMenuClear
mcMenuCopy
mcMenuCut
mcMenuPaste
mcMenuUndo
mcNotVisible
mcScaleMovieToFit`
mcTopLeftMovie
mcWithBadge
mcWithFrame
```

Document Revision History

This document supersedes "Movie Toolbox: Opening and Playing Movies."

This table describes the changes to *QuickTime Movie Playback Programming Guide*.

Date	Notes
2005-08-11	Major reorganization. Title changed from "Opening and Playing Movies."