

This chapter describes the Movie Toolbox and the key concepts that underlie QuickTime. The Movie Toolbox allows your application to use the full range of features provided by QuickTime. This toolbox provides functions that allow you to load, play, create, edit, and store objects that contain time-based data. If you are developing an application that works with time-based data, or if you are developing a component that will be used by movie applications, you should be familiar with the capabilities of the Movie Toolbox and the concepts discussed in this chapter.

This chapter is divided into the following major sections:

- “Introduction to Movies” discusses many of the concepts that are key to understanding how to use QuickTime, including time, movies, tracks, and media structures
- “About Movies” discusses the characteristics of QuickTime movies, tracks, and media structures
- “Using the Movie Toolbox” describes how you can use the Movie Toolbox to work with movies
- “Movie Toolbox Reference” describes the constants, data types, and functions provided by the Movie Toolbox
- “Summary of the Movie Toolbox” contains a condensed listing of the constants, data types, and functions provided by the Movie Toolbox in C and in Pascal

## Introduction to Movies

---

QuickTime allows you to manipulate time-based data such as video sequences, audio sequences, financial results from an ongoing business operation, laboratory data recorded over time, and so on. QuickTime uses the metaphor of a movie to describe time-based data. Therefore, QuickTime stores time-based data in objects called **movies**.

Just as a cinematic movie can contain several tracks (for example, a video track and a sound track), a single QuickTime movie can contain more than one stream of data. Following the movie metaphor, each of these data streams is called a *track*. Tracks in QuickTime movies do not actually contain the movie’s data. Rather, each track refers to a single media that, in turn, contains references to the actual media data. The media data may be stored on disks, CD-ROM volumes, videotape, or other appropriate storage devices.

Underlying all this is the notion of time. The next section describes how time is represented in QuickTime. Following that are sections that discuss how QuickTime movies, tracks, and media structures relate to time and to one another.

## Time and the Movie Toolbox

---

At the most basic level, the Movie Toolbox allows you to process time-based data. As such, the Movie Toolbox must provide a description of the time basis of that data as well as a definition of the context for evaluating that time basis. In QuickTime, a movie’s time

Movie Toolbox

basis is referred to as its **time base**. Geometrically, you can think of the time base as a vector that defines the direction and velocity of time for a movie. The context for a time base is called its **time coordinate system**. Essentially, the time coordinate system defines the axis on which the time base vector is plotted (see Figure 2-2 on page 2-8). The smallest single unit of time marked on that axis is defined by the time scale as the units per absolute second.

The following sections discuss each of these key concepts further.

Time Coordinate Systems

---

A movie’s time coordinate system provides the context for evaluating the passage of time in the movie. If you think of the time coordinate system as defining an axis for measuring time, it is only natural that this axis would be marked with a scale that defines a basic unit of measurement. In QuickTime, that measurement system is called a **time scale**.

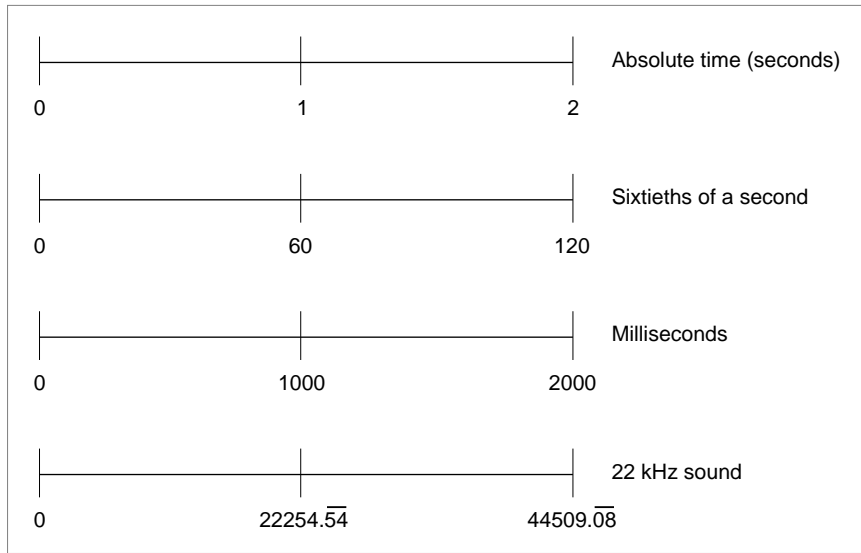
A QuickTime time scale defines the number of time units that pass each second in a given time coordinate system. A time coordinate system that has a time scale of 1 measures time in seconds. Similarly, a time coordinate system that has a time scale of 60 measures sixtieths of a second. In general, each time unit in a time coordinate system is equal to (1/time scale) seconds. Some common time scales are listed in Table 2-1.

**Table 2-1** Common movie time scales

---

Time scale	Absolute time measured
1	Seconds
60	Sixtieths of a second (Macintosh ticks)
1000	Milliseconds
22254. $\overline{54}$	Sound sampled at 22 kHz (kilohertz)

Figure 2-1 shows a duration of two seconds in absolute time and equivalent durations in the common time scales listed in Table 2-1.

**Figure 2-1** Time scales

A particular point in time in a time coordinate system is represented using a **time value**. A time value is expressed in terms of the time scale of its time coordinate system. Without an appropriate time scale, a time value is meaningless. For example, in a time coordinate system with a time scale of 60, a time value of 180 translates to 3 seconds. Because all time coordinate systems tie back to absolute time (that is, time as we measure it in seconds), the Movie Toolbox can translate time values from one time coordinate system into another.

Time coordinate systems have a finite maximum duration that defines the maximum time value for a time coordinate system (the minimum time value is always 0). Note that as a QuickTime movie is edited, the duration changes.

As the value of the time scale increases (as the time unit for a coordinate system gets smaller in terms of absolute time), the maximum absolute time that can be represented in a time coordinate system decreases. For example, if a time value were represented as an unsigned 16-bit integer, its maximum value would be 65,535. In a time coordinate system with a time scale of 1, the maximum time value would represent 65,535 seconds. However, in a time coordinate system with a time scale of 5, the maximum time value would correspond to 13,107 seconds. Hence, a time coordinate system's duration is limited by its time scale. QuickTime uses 32-bit and 64-bit quantities to represent time values, so you only need to worry about attaining a maximum absolute time in situations where a time coordinate system's duration is very long or its time scale is very large.

## Time Bases

A movie's time base defines its current time value and the **rate** at which time passes for the movie. The rate specifies the speed and direction in which time travels in a movie. Negative rate values cause you to move backward through a movie's data; positive values move forward. The time base also contains a reference to the clock that provides timing for the time base. QuickTime clocks are implemented as components that are managed by the Component Manager.

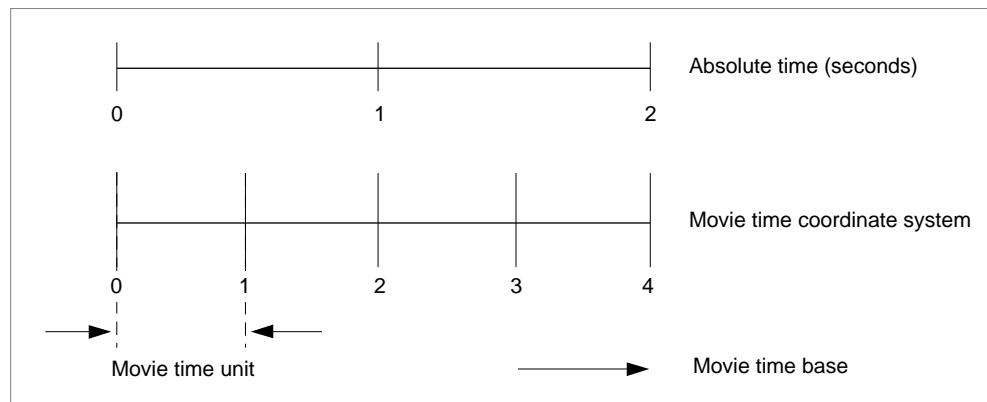
Time bases exist independently of any specific time coordinate system. However, time values extracted from a time base are meaningless without a time scale. Therefore, whenever you obtain a time value from a time base, you must specify the time scale of the time value result. The Movie Toolbox translates the time base's time value into a value that is sensible in the specified time scale.

### Note

A time base differs from a time coordinate system, which provides the foundation for a time base. (A time coordinate system is the field of play that defines the coordinate axis for a time base.) A time base operates in the context of a time coordinate system. It has a rate, which implies a direction as well as a speed through the movie. ♦

Figure 2-2 represents a time coordinate system and a time base geometrically. The time coordinate system is represented by a coordinate axis. In this example, the time coordinate system has a time scale of 2; that is, there are two time units in each second. The duration of this time coordinate system is 2 seconds, which is equivalent to 4 time units. An object's time base is depicted by the large arrow under the axis that represents the time coordinate system. This time base has a current time value of 3 and a rate of 1. The starting time is a time value, expressed in the units of the time coordinate system.

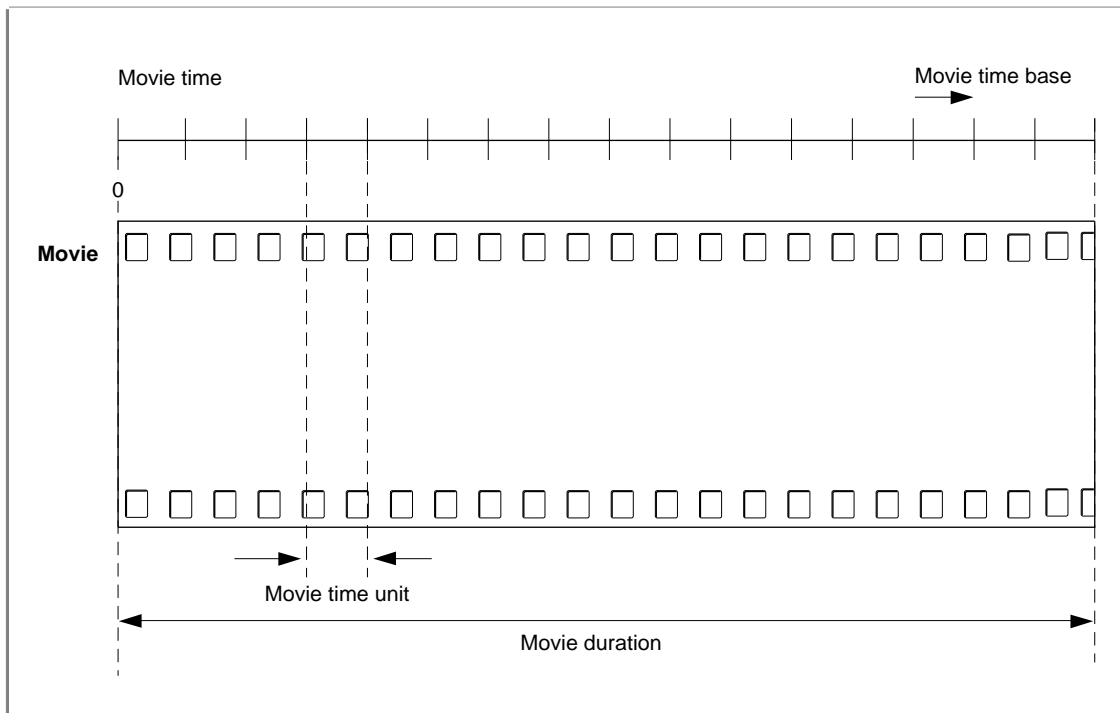
**Figure 2-2** A time coordinate system and a time base



## Movies

QuickTime movies have a time dimension defined by a time scale and a duration, which are specified by a time coordinate system. Figure 2-3 illustrates a movie's time coordinate system. A movie always starts at time 0. The time scale defines the unit of measure for the movie's time values. The **duration** specifies how long the movie lasts.

**Figure 2-3** A movie's time coordinate system



A movie can contain one or more tracks. Each track refers to media data that can be interpreted within the movie's time coordinate system. Each track begins at the beginning of the movie. However, a track can end at any time. In addition, the actual data in the track may be offset from the beginning of the movie. Tracks with data that does not commence at the beginning of a movie contain empty space that precedes the track data.

At any given point in time, one or more tracks may or may not be enabled.

### Note

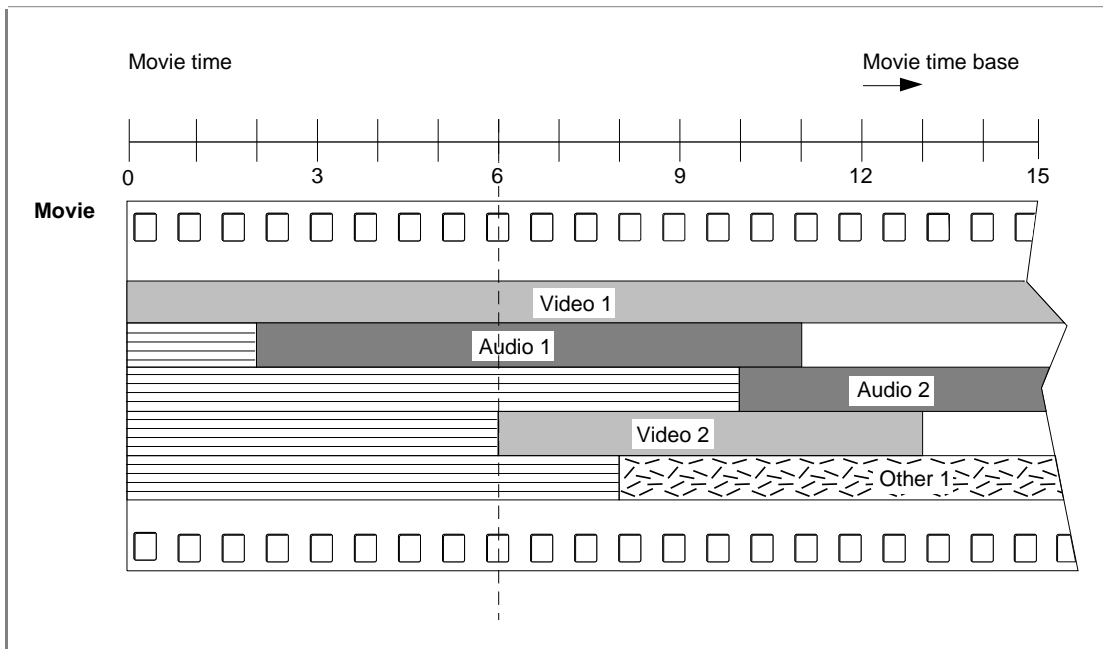
Throughout this book and its companion, *Inside Macintosh: QuickTime Components*, the term *enabled track* denotes a track that may become activated if the movie time intersects the track. An enabled track refers to a media that in turn refers to media data. ♦

## Movie Toolbox

However, no single track needs to be enabled during the entire movie. As you move through a movie, you gain access to the data that is described by each of the enabled tracks. Figure 2-4 shows a movie that contains five tracks. The lighter shading in each track represents the time offset between the beginning of the movie and the start of the track's data (this lighter shading corresponds to empty space at the beginning of these tracks). When the movie's time value is 6, there are three enabled tracks: Video 1 and Audio 1, and Video 2, which is just being enabled. The Other 1 track does not become enabled until the time value reaches 8. The Audio 2 track becomes enabled at time value 10.

A movie can contain one or more **layers**. Each layer contains one or more tracks that may be related to one another. The Movie Toolbox builds up a movie's visual representation layer by layer. For example, in Figure 2-4, if the images contained in the Video 1 and Video 2 tracks overlap spatially, the user sees the image that is stored in the front layer. You assign individual tracks to movie layers using Movie Toolbox functions that are described in "Working With Movie Spatial Characteristics" beginning on page 2-158.

**Figure 2-4** A movie containing several tracks



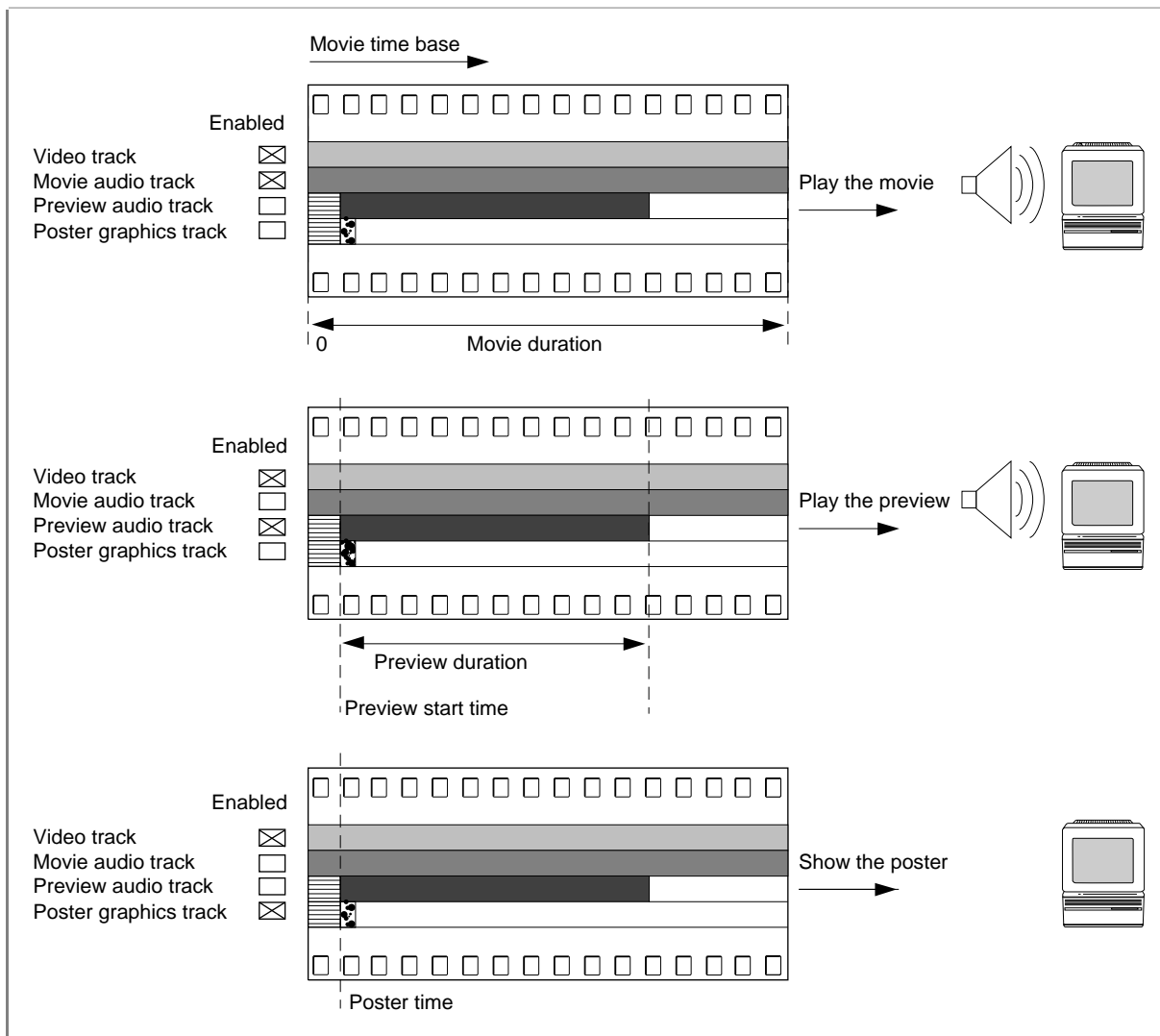
The Movie Toolbox allows you to define both a movie preview and a movie poster for a QuickTime movie. A **movie preview** is a short dynamic representation of a movie. Movie previews typically last no more than 3 to 5 seconds, and they should give the user some idea of what the movie contains. (An example of a movie preview is a narrative track.) You define a movie preview by specifying its start time, its duration, and its tracks. A movie may contain tracks that are used only in its preview.

## Movie Toolbox

A **movie poster** is a single visual image representing the movie. You specify a poster as a point in time in the movie. As with the movie itself and the movie preview, you define which tracks are enabled in the movie poster.

Figure 2-5 shows an example of a movie's tracks. The video track is used for the movie, the preview, and the poster. The movie audio track is used only for the movie. The preview audio track is used only for the preview. The poster graphics track is used only for the poster.

**Figure 2-5** A movie, its preview, and its poster

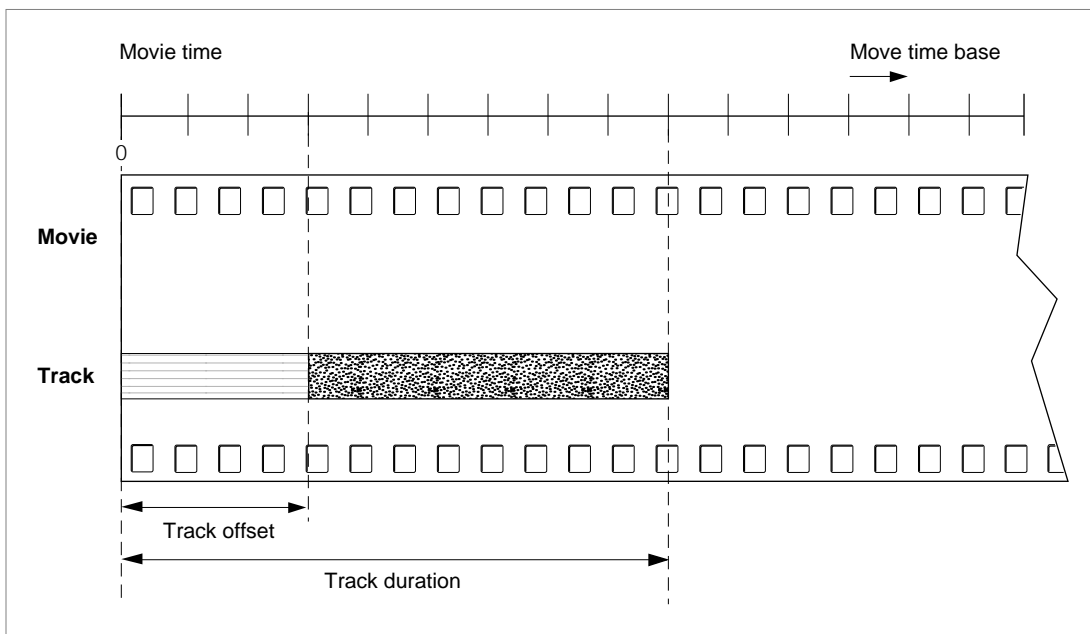


## Tracks

A movie can contain one or more tracks. Each track represents a single stream of data in a movie and is associated with a single media. The media has control information that refers to the actual movie data.

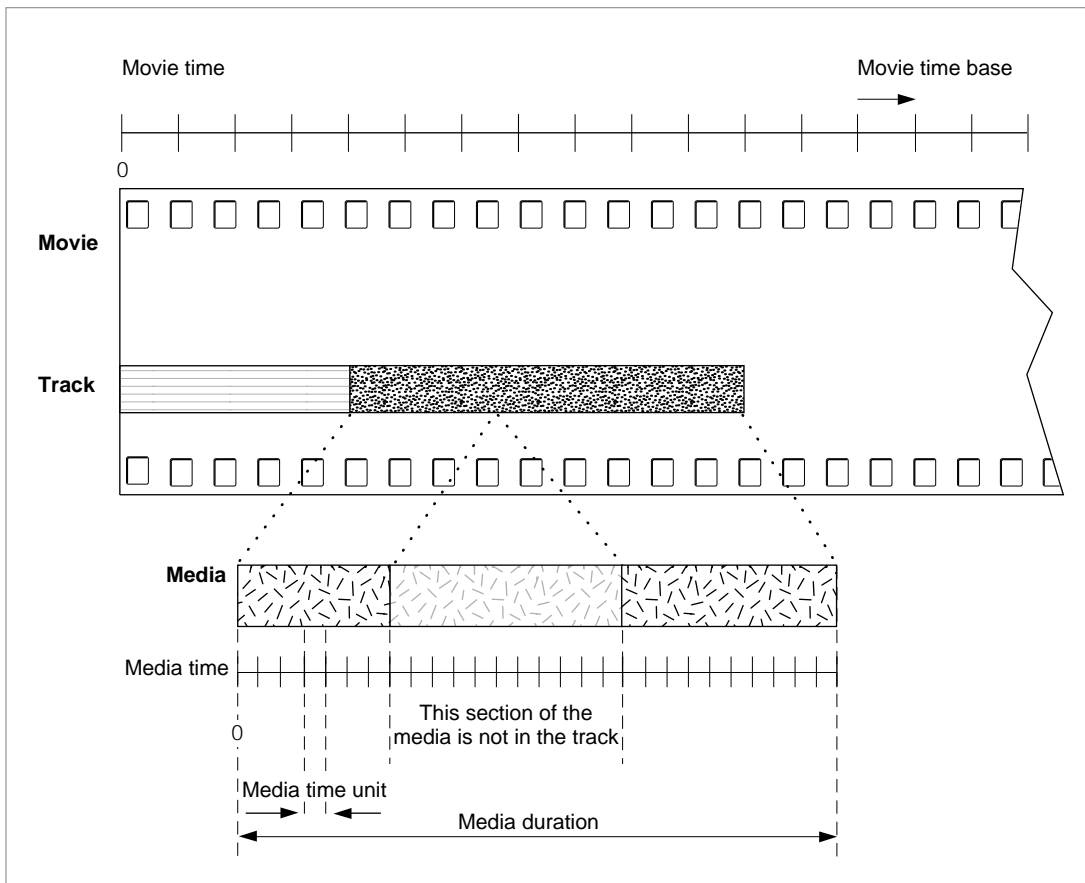
All of the tracks in a movie use the movie's time coordinate system. That is, the movie's time scale defines the basic time unit for each of the movie's tracks. Each track begins at the beginning of the movie, but the track's data might not begin until some time value other than 0. This intervening time is represented by blank space—in an audio track the blank space translates to silence; in a video track the blank space generates no visual image. Each track has its own duration. This duration need not correspond to the duration of the movie. Movie duration always equals the maximum duration of all the tracks. An example of this is shown in Figure 2-6.

**Figure 2-6** A track in a movie



A track is always associated with one media. The media contains control information that refers to the data that constitutes the track. The track contains a list of references that identify portions of the media that are used in the track. In essence, these references are an edit list of the media. Consequently, a track can play the data in its media in any order and any number of times. Figure 2-7 shows how a track maps data from a media into a movie.



**Figure 2-7** A track and its media

## Media Structures

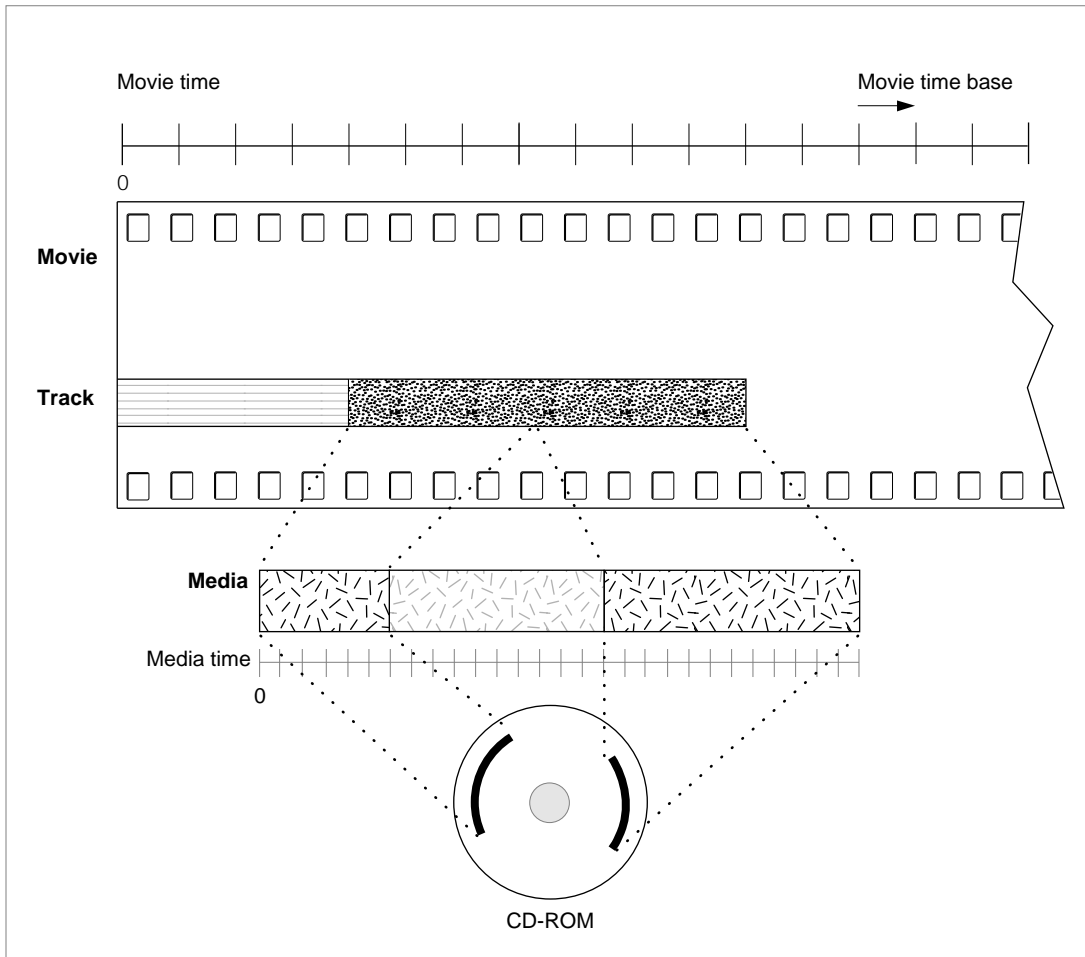
A media describes the data for a track. The data is not actually stored in the media. Rather, the media contains references to its media data, which may be stored in disk files, on CD-ROM discs, or other appropriate storage devices. Note that the data referred to by one media may be used by more than one movie, though the media itself is not reused.

Each media has its own time coordinate system, which defines the media's time scale and duration. A media's time coordinate system always starts at time 0, and it is independent of the time coordinate system of the movie that uses its data. Tracks map data from the movie's time coordinate system to the media's time coordinate system. Figure 2-7 shows how tracks perform this mapping.

Each supported data type has its own **media handler**. The media handler interprets the media's data. The media handler must be able to randomly access the data and play segments at rates specified by the movie. The track determines the order in which the media is played in the movie and maps movie time values to media time values.

Figure 2-8 shows the final link to the data. The media in the figure references digital video frames on a CD-ROM disc.

**Figure 2-8** A media and its data



## About Movies

This section discusses the characteristics that govern playing and storing movies, tracks, and media structures. This section has been divided into the following topics:

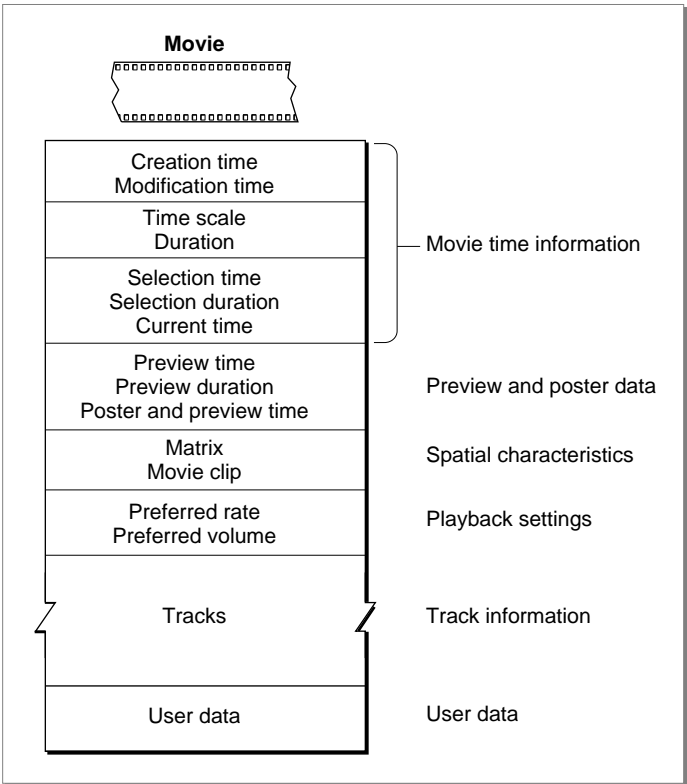
- "Movie Characteristics" discusses the time, display, and sound characteristics of a QuickTime movie
- "Track Characteristics" describes the characteristics of a movie track

- “Media Characteristics” discusses the characteristics of a media
- “Spatial Properties” describes how the Movie Toolbox displays a movie, including how the data from each media is collected and transformed prior to display
- “The Transformation Matrix” describes how matrix operations transform visual elements prior to display
- “Audio Properties” describes how the Movie Toolbox works with a movie’s sound tracks
- “Data Interchange” discusses how the format and content of a movie changes when it is stored on the scrap or in a file

## Movie Characteristics

A QuickTime movie is represented as a private data structure. Your application never works with individual fields in that data structure. Rather, the Movie Toolbox provides functions that allow you to work with a movie’s characteristics. Figure 2-9 shows some of the characteristics of a QuickTime movie.

**Figure 2-9**      Movie characteristics



## Movie Toolbox

Every QuickTime movie has some state information, including a creation time and a modification time. These times are expressed in standard Macintosh time format, representing the number of seconds since midnight, January 1, 1904. The creation time indicates when the movie was created. The modification time indicates when the movie was last modified and saved.

Each movie has its own time coordinate system and time scale. Any time values that relate to the movie must be defined using this time scale and must be between 0 and the movie's duration.

A movie's preview is defined by its starting time and duration. Both of these time values are expressed in terms of the movie's time scale. A movie's poster is defined by its time value, which is in terms of the movie's time scale. You assign tracks to the movie preview and the movie poster by calling the Movie Toolbox functions that are described later in this chapter.

Your current position in a movie is defined by the movie's **current time**. If the movie is currently playing, this time value is changing. When you save a movie in a movie file, the Movie Toolbox updates the movie's current time to reflect its current position. When you load a movie from a movie file, the Movie Toolbox sets the movie's current time to the value found in the movie file.

The Movie Toolbox provides high-level editing functions that work with a movie's **current selection**. The current selection defines a segment of the movie by specifying a start time, referred to as the **selection time**, and a duration, called the **selection duration**. These time values are expressed using the movie's time scale.

For each movie currently in use, the Movie Toolbox maintains an **active movie segment**. The active movie segment is the part of the movie that your application is interested in playing. By default, the active movie segment is set to be the entire movie. You may wish to change this to be some segment of the movie—for example, if you wish to play a user's selection repeatedly. By setting the active movie segment, you guarantee that the Movie Toolbox uses no samples from outside of that range while playing the movie. See "Enhancing Movie Playback Performance," which begins on page 2-134, for details on functions that work with the active segment.

A movie's display characteristics are specified by a number of elements. The movie has a movie clipping region and a 3-by-3 transformation matrix. The Movie Toolbox uses these elements to determine the spatial characteristics of the movie. See "Spatial Properties" beginning on page 2-20 for a complete description of these elements and how they are used by the Movie Toolbox.

When you save a movie, you can establish preferred settings for playback rate and volume. The preferred playback rate is called the **preferred rate**. The preferred playback volume is called the **preferred volume**. These settings represent the most natural values for these movie characteristics. When the Movie Toolbox loads a movie from a movie file, it sets the movie's volume to this preferred value. When you start playing the movie, the Movie Toolbox uses the preferred rate. You can then use Movie Toolbox functions to change the rate and volume during playback.

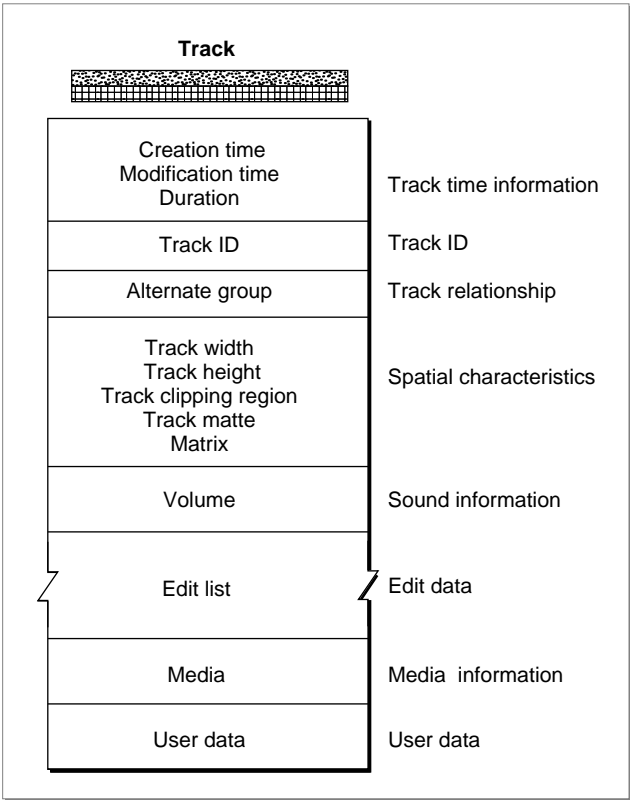
Movies contain each of their tracks. See the next section for more information about tracks and their characteristics.

The Movie Toolbox allows your application to store its own data along with a movie. You define the format and content of these data objects. This application-specific data is called **user data**. You can use these data objects to store both text and binary data. For example, you can use text user data items to store a movie’s copyright and credit information. The Movie Toolbox provides functions that allow you to set and retrieve a movie’s user data. This data is saved with the movie when you save the movie.

## Track Characteristics

A QuickTime track is represented as a private data structure. Your application never works with individual fields in that data structure. Rather, the Movie Toolbox provides functions that allow you to work with a track’s characteristics. Figure 2-10 shows the characteristics of a QuickTime track.

**Figure 2-10** Track characteristics



As with movies, each track has some state information, including a creation time and a modification time. These times are expressed in standard Macintosh time format, representing the number of seconds since midnight, January 1, 1904. The creation time

## Movie Toolbox

indicates when the track was created. The modification time indicates when the track was last modified and saved.

Each track has its own duration value, which is expressed in the time scale of the movie that contains the track.

As has been discussed, movies can contain more than one track. In fact, a movie can contain more than one track of a given type. You might want to create a movie with several sound tracks, each in a different language, and then activate the sound track that is appropriate to the user's native language. Your application can manage these collections of tracks by assigning each track of a given type to an **alternate group**. You can then choose one track from that group to be enabled at any given time. You can select a track from an alternate group based on its language or its **playback quality**. A track's playback quality indicates its suitability for playback in a given environment. All tracks in an alternate group should refer to the same type of data.

A track's display characteristics are specified by a number of elements, including track width, track height, a transformation matrix, and a clipping region. See "Spatial Properties," which begins on page 2-20, for a complete description of these elements and how they are used by the Movie Toolbox.

Each track has a current volume setting. This value controls how loudly the track plays relative to the movie volume.

Perhaps most important, tracks contain a media edit list. The edit list contains entries that define how the track's media is to be used in the movie that contains the track. Each entry in the edit list indicates the starting time and duration of the media segment, along with the playback rate for that segment.

Each track contains its associated media. See the next section for more information about media structures and their characteristics.

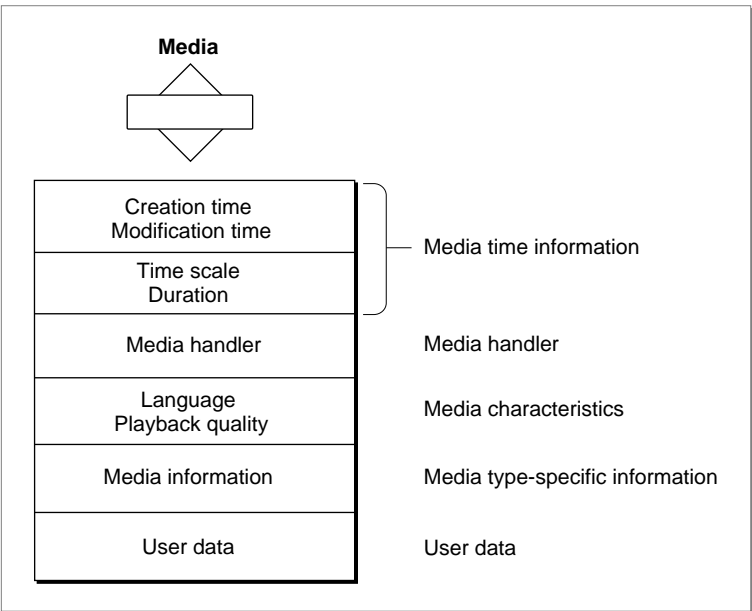
The Movie Toolbox allows your application to store its own user data along with a track. You define the format and content of these data objects. The Movie Toolbox provides functions that allow you to set and retrieve a track's user data. This data is saved with the track when you save the movie.

## Media Characteristics

---

As is the case with movies and tracks, a QuickTime media is represented as a private data structure. Your application never works with individual fields in that data structure. Rather, the Movie Toolbox provides functions that allow you to work with a media's characteristics. Figure 2-11 shows the characteristics of a QuickTime media.

Figure 2-11 Media characteristics



Each QuickTime media has some state information, including a creation time and a modification time. These times are expressed in standard Macintosh time format, representing the number of seconds since midnight, January 1, 1904. The creation time indicates when the media was created. The modification time indicates when the media was last modified and saved.

Each media has its own time coordinate system, which is defined by its time scale and duration. Any time values that relate to the media must be defined in terms of this time scale and must be between 0 and the media's duration.

A media contains information that identifies its language and playback quality. These values are used when selecting from among the tracks in an alternate group.

The media specifies a media handler, which is responsible for the details of loading, storing, and playing media data. The media handler can store state information in the media. This information is referred to as **media information**. The media information identifies where the media's data is stored and how to interpret that data. Typically, this data is stored in a **data reference**, which identifies the file that contains the data and the type of data that is stored in the file.

The Movie Toolbox allows your application to store its own user data along with a media. You define the format and content of these data objects. The Movie Toolbox provides functions that allow you to set and retrieve a media's user data. This data is saved with the media when you save the movie.

## Spatial Properties

---

When you play a movie that contains visual data, the Movie Toolbox gathers the movie's data from the appropriate tracks and media structures, transforms the data as appropriate, and displays the results in a window. The Movie Toolbox uses only those tracks that

- are not empty
- contain media structures that reference data at a specified time
- are enabled in the current movie mode (standard playback, poster mode, or preview mode)

Consequently, the size, shape, and location of many of these regions may change during movie playback. This process is quite complicated and involves several phases of clipping and resizing.

The Movie Toolbox shields you from the intricacies of this process by providing two high-level functions, `GetMovieBox` and `SetMovieBox` (described on page 2-162 and page 2-161, respectively), which allow you to place a movie box at a specific location in the display coordinate system. When you use these functions, the Movie Toolbox automatically adjusts the contents of the movie's matrix to satisfy your request.

Figure 2-12 provides an overview of the entire process of gathering, transforming, and displaying visual data. Each track defines its own spatial characteristics, which are then interpreted within the context of the movie's spatial characteristics.

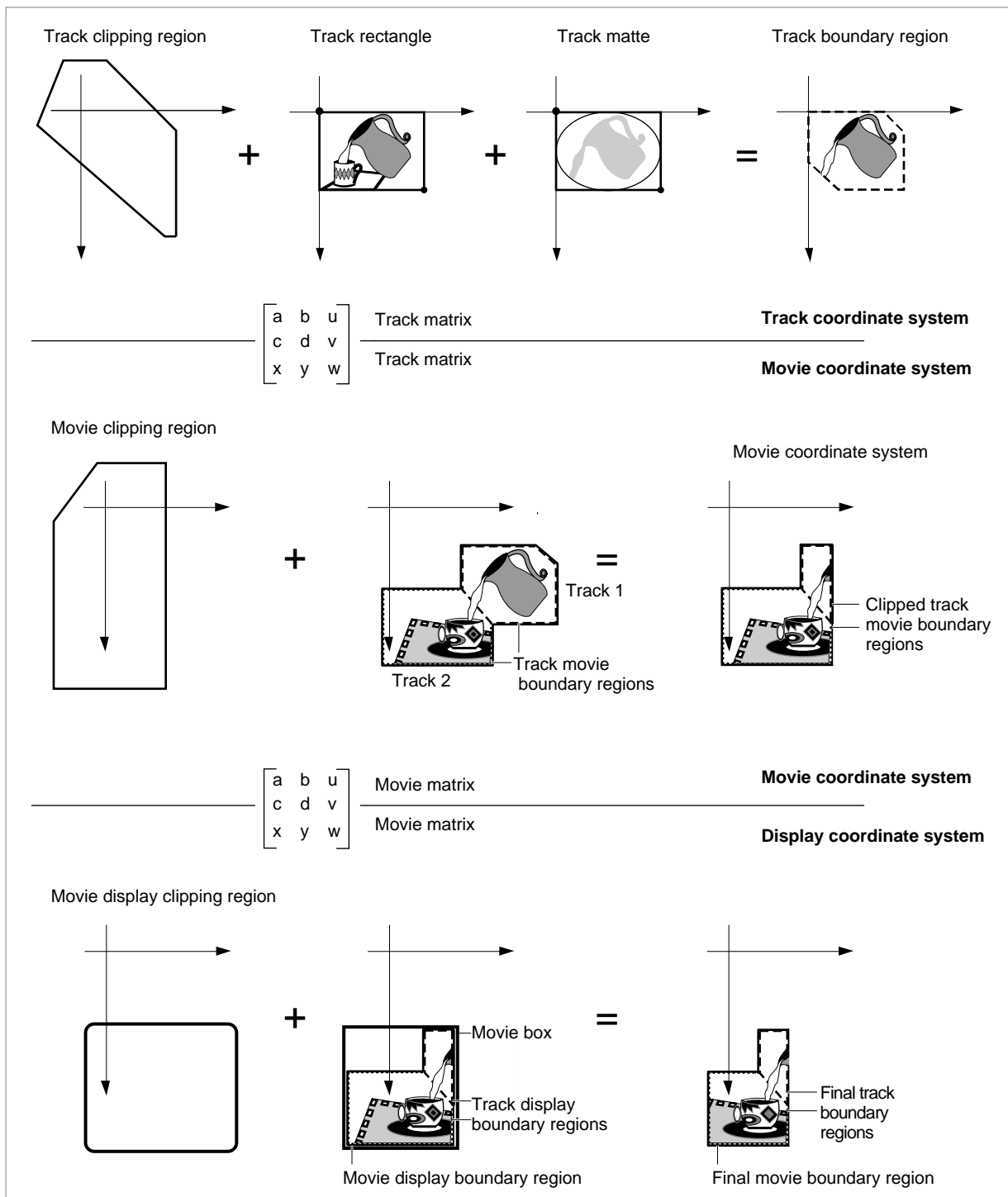
This section describes the process that the Movie Toolbox uses to display a movie. The process begins with the movie data and ends with the final movie display. The phases, which are described in detail in this section, include

1. the creation of a track rectangle (see Figure 2-13 on page 2-22)
2. the clipping of a track's image (see Figure 2-14 on page 2-23)
3. the transformation of a track into the movie coordinate system (see Figure 2-15 on page 2-23)
4. the clipping of a movie image (see Figure 2-16 on page 2-24)
5. the transformation of a movie into the display coordinate system (see Figure 2-17 on page 2-25)
6. the clipping of a movie for final display (see Figure 2-18 on page 2-25)

### Note

Throughout this book and in *Inside Macintosh: QuickTime Components*, the term *time coordinate system* denotes QuickTime's time-based system. All other instances of the term *coordinate system* refer to QuickDraw's graphic coordinates. ♦



**Figure 2-12** Spatial processing of a movie and its tracks

## Movie Toolbox

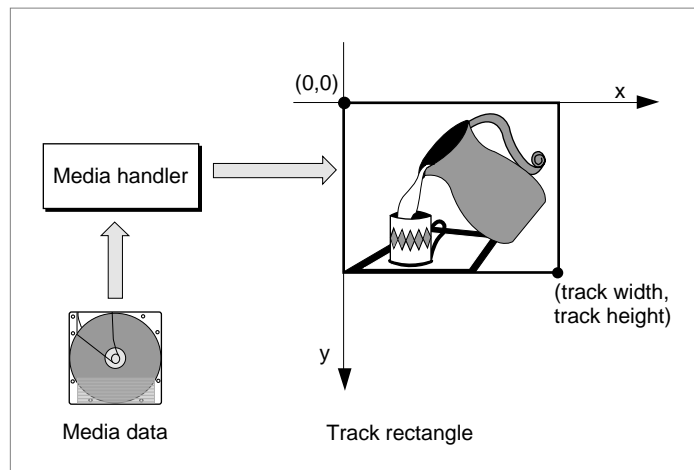
Each track defines a rectangle into which its media is displayed. This rectangle is referred to as the **track rectangle**, and it is defined by the **track width** and **track height** values assigned to the track. The upper-left corner of this rectangle defines the origin point of the track's *coordinate system*.

**Note**

Henceforth, the graphic coordinate system for a track is referred to simply as its *coordinate system*. ♦

The media handler associated with the track's media is responsible for displaying an image into this rectangle. This process is shown in Figure 2-13.

**Figure 2-13** A track rectangle



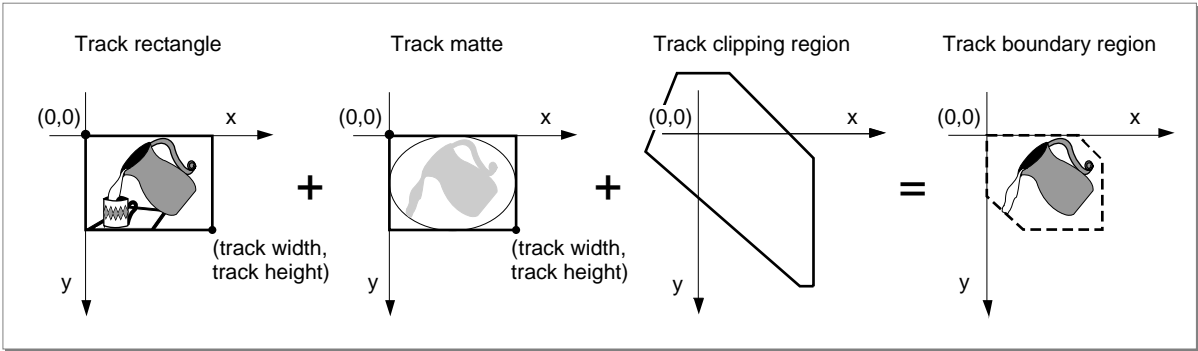
The Movie Toolbox next mattes the image in the track rectangle by applying the track matte and the track clipping region. This does not affect the shape of the image—only the display. Both the track matte and the track clipping region are optional.

A **track matte** provides a mechanism for mixing images. Mattes contain several bits per pixel and are defined in the track's coordinate system. The matte can be used to perform a deep-mask operation on the image in the track rectangle. The Movie Toolbox displays the weighted average of the track and its destination based on the corresponding pixel value in the matte.

The **track clipping region** is a QuickDraw region that defines a portion of the track rectangle to retain. The track clipping region is defined in the track's coordinate system. This clipping operation creates the **track boundary region**, which is the intersection of the track rectangle and the track clipping region.

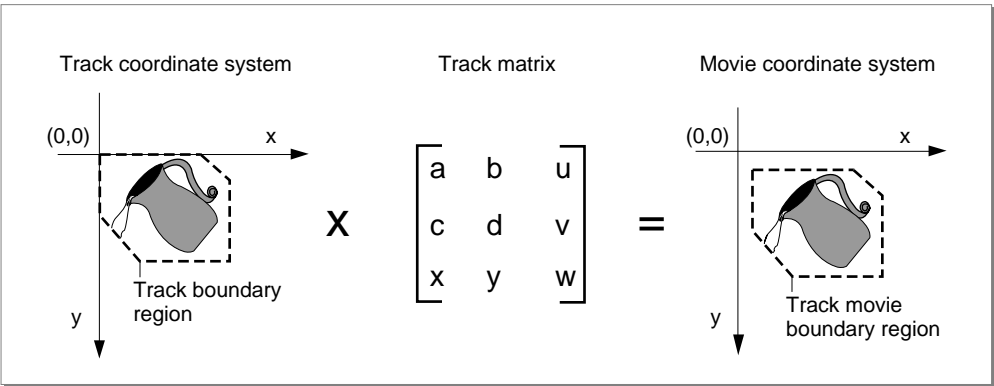
This process and its results are shown in Figure 2-14.

**Figure 2-14** Clipping a track's image



After clipping and matting the track's image, the Movie Toolbox transforms the resulting image into the movie's coordinate system. The Movie Toolbox uses a 3-by-3 transformation matrix to accomplish this operation (see the next section, "The Transformation Matrix," for a complete discussion of matrix operations in the Movie Toolbox). The image inside the track boundary region is transformed by the track's matrix into the movie coordinate system. The resulting area is bounded by the **track movie boundary region**. Figure 2-15 shows the results of this transformation operation.

**Figure 2-15** A track transformed into a movie coordinate system

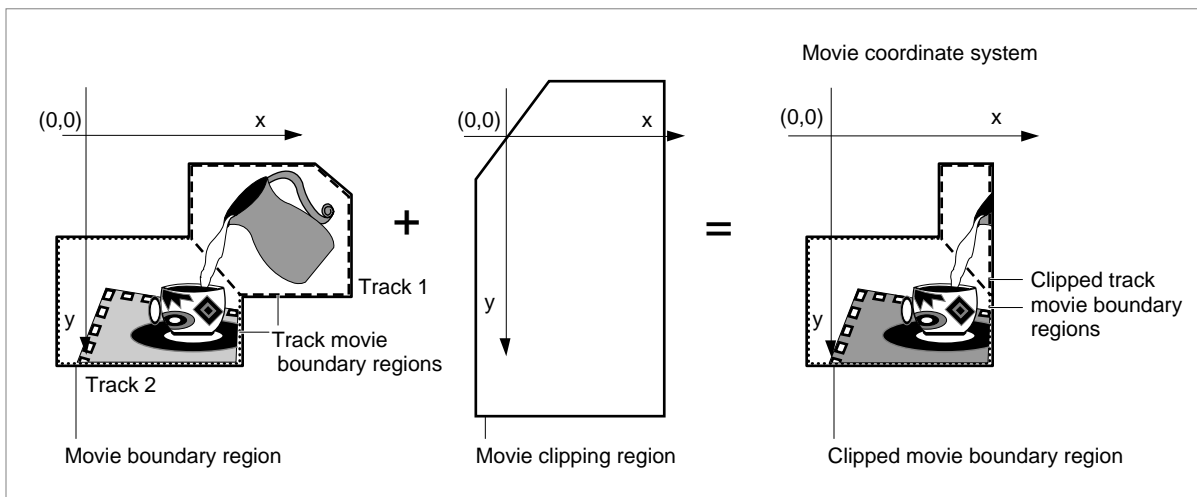


## Movie Toolbox

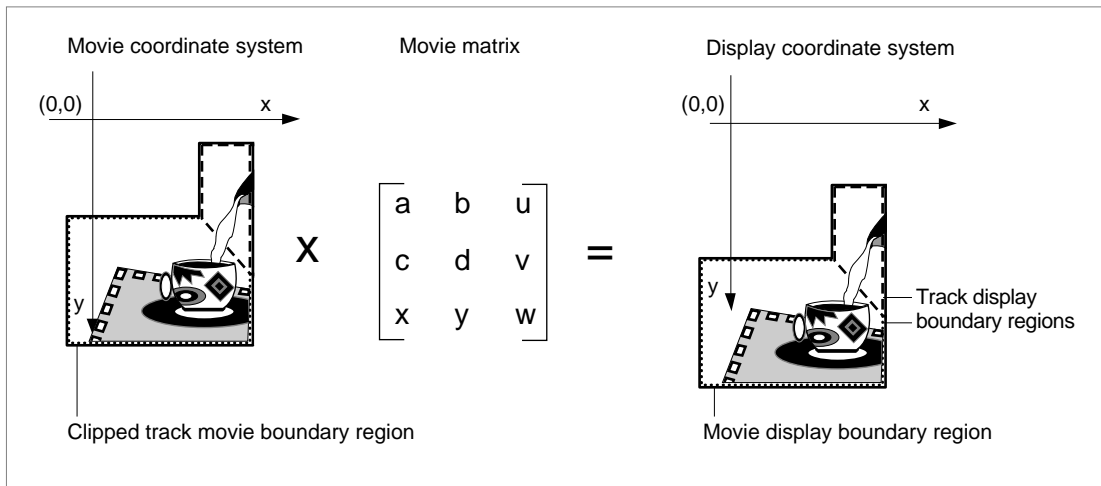
The Movie Toolbox performs this portion of the process for each track in the movie. Once all of the movie's tracks have been processed, the Movie Toolbox proceeds to transform the complete movie image for display.

The union of all track movie boundary regions for a movie defines the movie's **movie boundary region**. The Movie Toolbox combines a movie's tracks into this single region where layers are applied. Therefore, tracks in back layers may be partially or completely obscured by tracks in front layers. The Movie Toolbox clips this region to obtain the **clipped movie boundary region**. The movie's **movie clipping region** defines the portion of the movie boundary region that is to be used. Figure 2-16 shows the process by which a movie is clipped and the resulting clipped movie boundary region.

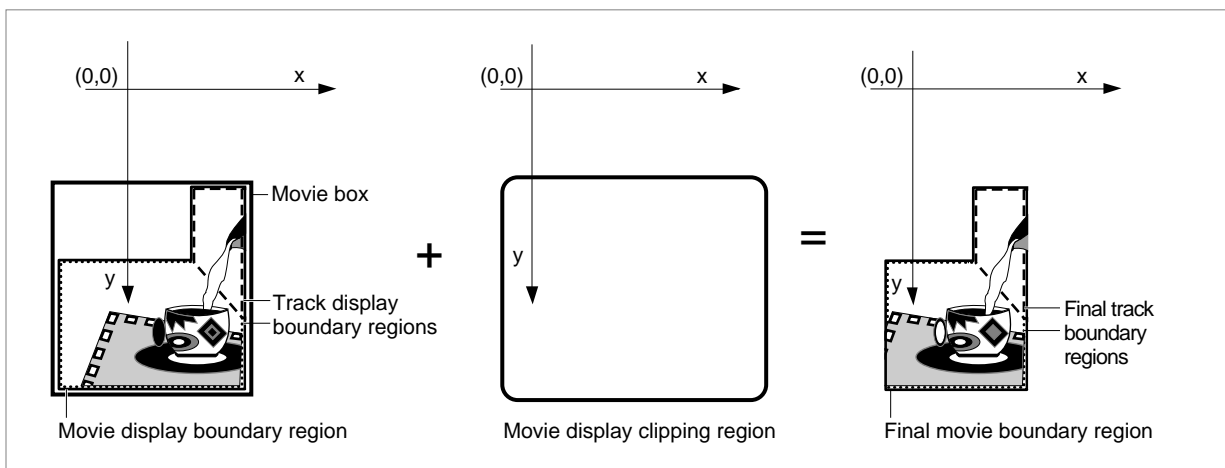
**Figure 2-16** Clipping a movie's image



After clipping the movie's image, the Movie Toolbox transforms the resulting image into the display coordinate system. The Movie Toolbox uses a 3-by-3 transformation matrix to accomplish this operation (see the next section, "The Transformation Matrix," for a complete discussion of matrix operations in the Movie Toolbox). The image inside the clipped movie boundary region is transformed by the movie's matrix into the display coordinate system. The resulting area is bounded by the movie display boundary region. Figure 2-17 shows the results of this step.

**Figure 2-17** A movie transformed to the display coordinate system

The rectangle that encloses the movie display boundary region is called the **movie box**, as shown in Figure 2-18. You can control the location of a movie's movie box by adjusting the movie's transformation matrix.

**Figure 2-18** Clipping a movie for final display

Once the movie is in the **display coordinate system** (that is, the QuickDraw graphics world), the Movie Toolbox performs a final clipping operation to generate the image that is displayed. The movie is clipped with the **movie display clipping region**. When a movie is displayed, the Movie Toolbox ignores the graphics port's clipping region—this is why there is a movie display clipping region. Figure 2-18 shows this operation.

## The Transformation Matrix

The Movie Toolbox makes extensive use of transformation matrices to define graphical operations that are performed on movies when they are displayed. A **transformation matrix** defines how to map points from one coordinate space into another coordinate space. By modifying the contents of a transformation matrix, you can perform several standard graphical display operations, including translation, rotation, and scaling. The Movie Toolbox provides a set of functions that make it easy for you to manipulate translation matrices. Those functions are discussed in “Matrix Functions” which begins on page 2-341. The remainder of this section provides an introduction to matrix operations in a graphical environment.

The matrix used to accomplish two-dimensional transformations is described mathematically by a 3-by-3 matrix. Figure 2-19 shows a sample 3-by-3 matrix. Note that QuickTime assumes that the values of the matrix elements *u* and *v* are always 0.0, and the value of matrix element *w* is always 1.0.

**Figure 2-19** A point transformed by a 3-by-3 matrix

$$\begin{bmatrix} x & y & 1 \end{bmatrix} \times \begin{bmatrix} a & b & u \\ c & d & v \\ t_x & t_y & w \end{bmatrix} = \begin{bmatrix} x' & y' & 1 \end{bmatrix}$$

During display operations, the contents of a 3-by-3 matrix transform a point (*x,y*) into a point (*x',y'*) by means of the following equations:

$$x' = ax + cy + t_x$$

$$y' = bx + dy + t_y$$

For example, the matrix shown in Figure 2-20 performs no transformation. It is referred to as the **identity matrix**.

**Figure 2-20** The identity matrix

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

## Movie Toolbox

Using the formulas discussed earlier, you can see that this matrix would generate a new point (x',y') that is the same as the old point (x,y):

$$x' = 1x + 0y + 0$$

$$y' = 0x + 1y + 0$$

$$x' = y \text{ and } y' = x$$

In order to move an image by a specified displacement, you perform a translation operation. This operation modifies the x and y coordinates of each point by a specified amount. The matrix shown in Figure 2-21 describes a translation operation.

**Figure 2-21** A matrix that describes a translation operation

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

You can stretch or shrink an image by performing a scaling operation. This operation modifies the x and y coordinates by some factor. The magnitude of the x and y factors governs whether the new image is larger or smaller than the original. In addition, by making the x factor negative, you can flip the image about the x-axis; similarly, you can flip the image horizontally, about the y-axis, by making the y factor negative. The matrix shown in Figure 2-22 describes a scaling operation.

**Figure 2-22** A matrix that describes a scaling operation

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

## Movie Toolbox

Finally, you can rotate an image by a specified angle by performing a rotation operation. You specify the magnitude and direction of the rotation by specifying factors for both  $x$  and  $y$ . The matrix shown in Figure 2-23 rotates an image counterclockwise by an angle  $\theta$ .

**Figure 2-23** A matrix that describes a rotation operation

$$\begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 \\ -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

You can combine matrices that define different transformations into a single matrix. The resulting matrix retains the attributes of both transformations. For example, you can both scale and translate an image by defining a matrix similar to that shown in Figure 2-24.

**Figure 2-24** A matrix that describes a scaling and translation operation

$$\begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ t_x & t_y & 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ t_x & t_y & 1 \end{bmatrix}$$

You combine two matrices by concatenating them. Mathematically, the two matrices are combined by matrix multiplication. Note that the order in which you concatenate matrices is important—matrix operations are not commutative.

Transformation matrices used by the Movie Toolbox contain the following data types:

```
[0] [0]Fixed    [1] [0]Fixed    [2] [0]Fract
[0] [1]Fixed    [1] [1]Fixed    [2] [1]Fract
[0] [2]Fixed    [1] [2]Fixed    [2] [2]Fract
```

Each cell in this table represents the data type of the corresponding element of a 3-by-3 matrix. All of the elements in the first two columns of a matrix are represented by `Fixed` values. Values in the third column are represented as `Fract` values. The `Fract` data type specifies a 32-bit, fixed-point value that contains 2 integer bits and 30 fractional bits. This data type is useful for accurately representing numbers in the range from  $-2$  to  $2$ . For more information about the `Fract` data type, see *Inside Macintosh: Imaging*.



## Audio Properties

---

This section discusses the sound capabilities of QuickTime and the Movie Toolbox. It has been divided into the following topics:

- “Sound Playback” discusses the playback capabilities of the Movie Toolbox
- “Adding Sound to Video” discusses several issues you should consider when creating movies that contain both sound and video
- “Sound Data Formats” describes the formats the Movie Toolbox uses to store sound information

### Sound Playback

---

As is the case with video data, QuickTime movies store sound information in tracks. QuickTime movies may have one or more sound tracks. The Movie Toolbox can play more than one sound at a time by mixing the enabled sound tracks together during playback. This allows you to put together movies with separate music and voice tracks. You can then manipulate the tracks separately but play them together. You can also use multiple sound tracks to store different languages.

There are two main attributes of sound in QuickTime movies: volume and balance. You can control these attributes using the facilities of the Movie Toolbox.

Every QuickTime movie has a current volume setting. This volume setting controls the loudness of the movie’s sound. You can adjust a movie’s current volume by calling the `SetMovieVolume` function (described on page 2-182). In addition, you can set a preferred volume setting for a movie. This value represents the best volume for the movie. The Movie Toolbox saves this value when you store a movie into a movie file. The value of the current volume is lost. You can set a movie’s preferred volume by calling the `SetMoviePreferredVolume` function (described on page 2-132). When you load a movie from a movie file, the Movie Toolbox sets the movie’s current volume to the value of its preferred volume.

Each track in a movie also has a volume setting. A track’s volume governs its loudness relative to other tracks in the movie. You can set a track’s volume by calling the `SetTrackVolume` function (described on page 2-183).

In the Movie Toolbox, movie and track volumes are represented as 16-bit, fixed-point numbers that range from  $-1.0$  to  $+1.0$ . The high-order 8 bits contain the integer portion of the value; the low-order 8 bits contain the fractional part. Positive values denote volume settings, with  $1.0$  corresponding to the maximum volume on your computer. Negative values are muted, but retain the magnitude of the volume setting so that, by toggling the sign of a volume setting, you can turn off the sound and then turn it back on at the previous level (something like pressing the mute button on a radio).

A track’s volume is scaled to a movie’s volume, and the movie’s volume is scaled to the value the user specifies for speaker volume using the Sound control panel. That is, a movie’s volume setting represents the maximum loudness of any track in the movie. If you set a track’s volume to a value less than  $1.0$ , that track plays proportionally quieter, relative to the loudness of other tracks in the movie.

## Movie Toolbox

Each track in a movie has its own balance setting. The balance setting controls the mix of sound between a computer's two speakers. If the source sound is monaural, the balance setting controls the relative loudness of each speaker. If the source sound is stereo, the balance setting governs the mix of the right and left channels. You can set the balance for a track's media by calling the `SetSoundMediaBalance` function (described on page 2-289). When you save the movie, the balance setting is stored in the movie file.

In the Movie Toolbox, balance values are represented as 16-bit, fixed-point numbers that range from  $-1.0$  to  $+1.0$ . The high-order 8 bits contain the integer portion of the value; the low-order 8 bits contain the fractional part. Negative values weight the balance toward the left speaker; positive values emphasize the left channel. Setting the balance to 0 corresponds to a neutral setting.

## Adding Sound to Video

---

Most QuickTime movies contain both sound data and video data. If you are creating an application that plays movies, you do not need to worry about the details of how sound is stored in a movie. However, if you are developing an application that creates movies, you need to consider how you store the sound and video data.

There are two ways to store sound data in a QuickTime movie. The simplest method is to store the sound track as a continuous stream. When you play a movie that has its sound in this form, the Movie Toolbox loads the entire sound track into memory, and then reads the video frames when they are needed for display. While this technique is very efficient, it requires a large amount of memory to store the entire sound, which limits the length of the movie. This technique also requires a large amount of time to read in the entire sound track before the movie can start playing. For this reason, this technique is only recommended when the sound for a movie is fairly small (less than 64 KB).

For larger movies, a technique called **interleaving** must be used so that the sound and video data may be alternated in small pieces, and the data can be read off disk as it is needed. Interleaving allows for movies of almost any length with little delay on startup. However, you must tune the storage parameters to avoid a lower video frame rate and breaks in the sound that result when sound data is read from slow storage devices. In general, the Movie Toolbox hides the details of interleaving from your application. The `FlattenMovie` and `FlattenMovieData` functions (described on page 2-105 and page 2-107, respectively) allow you to enable and disable interleaving when you create a movie. These functions then interact with the appropriate media handler to correctly interleave the sound and video data for your movie. For more information about working with sound, see the chapter "Sound Manager" in *Inside Macintosh: More Macintosh Toolbox*.

## Sound Data Formats

The Movie Toolbox stores sound data in sound tracks as a series of digital samples. Each sample specifies the amplitude of the sound at a given point in time, a format commonly known as *linear pulse-code modulation* (linear PCM). The Movie Toolbox supports both monaural and stereo sound. For monaural sounds, the samples are stored sequentially, one after another. For stereo sounds, the samples are stored interleaved in a left/right/left/right fashion.

In order to support a broad range of audio data formats, the Movie Toolbox can accommodate a number of different sample encoding formats, sample sizes, sample rates, and compression algorithms. The following paragraphs discuss the details of each of these attributes of movie sound data.

The Movie Toolbox supports two techniques for encoding the amplitude values in a sample: offset-binary and twos-complement. **Offset-binary encoding** represents the range of amplitude values as an unsigned number, with the midpoint of the range representing silence. For example, an 8-bit sample stored in offset-binary format would contain sample values ranging from 0 to 255, with a value of 128 specifying silence (no amplitude). Samples in Macintosh sound resources are stored in offset-binary form.

**Twos-complement encoding** stores the amplitude values as a signed number—in this case silence is represented by a sample value of 0. Using the same 8-bit example, twos-complement values would range from -128 to 127, with 0 meaning silence. The Audio Interchange File Format (AIFF) used by the Sound Manager stores samples in twos-complement form, so it is common to see this type of sound in QuickTime movies.

The Movie Toolbox allows you to store information about the sound data in the sound description. See “The Sound Description Structure,” which begins on page 2-79, for details on the sound description structure. Sample size indicates the number of bits used to encode the amplitude value for each sample. The size of a sample determines the quality of the sound, since more bits can represent more amplitude values. The basic Macintosh sound hardware supports only 8-bit samples, but the Sound Manager also supports 16-bit and 32-bit sample sizes. The Movie Toolbox plays these larger samples on 8-bit Macintosh hardware by converting the samples to 8-bit format before playing them.

Sample rate indicates the number of samples captured per second. The sample rate also influences the sound quality, because higher rates can more accurately capture the original sound waveform. The basic Macintosh hardware supports an output sampling rate of 22.254 kHz. The Movie Toolbox can support any rate up to 65.535 kHz; as with sample size, the Movie Toolbox converts higher sample rates to rates that can be accommodated by the Macintosh hardware when it plays the sound.

In addition to these sample encoding formats, the Movie Toolbox also supports the Macintosh Audio Compression and Expansion (MACE) capability of the Sound Manager. This allows compression of the sound data at ratios of 3 to 1 or 6 to 1. Compressing a movie’s sound can yield significant savings in storage and RAM space, at the cost of somewhat lower quality and higher CPU overhead on playback.

## Data Interchange

---

This section discusses how you can exchange movies between applications on your Macintosh computer or between your Macintosh and other computers.

### Movies on the Clipboard

---

Working with QuickTime and applications that employ QuickTime, the user may cut, copy, and paste movies just like any other type of data. When your application performs a cut or a copy operation, the Movie Toolbox returns a movie. Use the Movie Toolbox's `PutMovieOnScrap` and `NewMovieFromScrap` functions (described on page 2-244 and page 2-245, respectively) to work with movies on the scrap.

Because a movie contains only references to its media data, it is small enough to put onto the scrap.

### Movies in Files

---

A QuickTime movie file typically stores a movie in the resource fork of the file. The data for this movie may reside in the data fork of the same file, or in other files. In fact, a movie file may have no data fork at all—all the data for a movie may reside in other files. This allows several movies to share the same data.

The data referenced by a media is always stored in the data fork of a file. Because a movie can contain more than one media, and each media in a movie can refer to a different data file, it follows that a single movie may refer to more than one data file.

The Movie Toolbox allows you to create a movie file that contains all of its movie data. Such files are called *self-contained movie files*. Self-contained movie files can be used to move a movie from one Macintosh computer to another.

The Movie Toolbox also accommodates operating systems that do not recognize files with more than one fork. In this case, you can create a movie file that stores the movie and all of its data in the data fork of the Macintosh file. You can then transfer that file to a computer that runs another operating system. For more information, see the chapter “Movie Resource Formats” later in this book.

## Using the Movie Toolbox

---

The Movie Toolbox provides functions that allow applications to control all aspects of movies in Macintosh computer applications. There are Movie Toolbox functions that provide basic operations for opening and playing movies as well as more complex functions for the creation and manipulation of the data that makes up the movie's media.

## Movie Toolbox

This section discusses a number of the more common operations your application may perform with the Movie Toolbox, and it has been divided into the following sections:

- “Determining Whether the Movie Toolbox Is Installed” describes how to use the Gestalt Manager to retrieve the version of the Movie Toolbox that is installed
- “Getting Ready to Work With Movies” describes the steps you must take before you can work with QuickTime movies
- “Getting a Movie From a File” discusses how to load a movie from a movie file
- “Playing Movies With a Movie Controller” shows how you can use a movie controller component to simplify playing a movie
- “Playing a Movie” describes how to play a movie using Movie Toolbox functions
- “Movies and the Scrap” discusses how your application can place movies onto the system scrap and retrieve movies from the scrap
- “Creating a Movie” shows how you can create a new movie
- “Saving Movies in Movie Files” describes how to save movies into movie files
- “Using Movies in Your Event Loop” discusses how to grant time to the Movie Toolbox to allow your movies to play
- “The Movie Toolbox and System 6” discusses using the Movie Toolbox on Macintosh computers that are running System 6
- “Previewing Files” describes how to create and display file previews
- “Using Application-Defined Functions” describes how your application can retrieve information about long Movie Toolbox operations and perform custom display processing
- “Working With Movie Spatial Characteristics” shows how to create a track matte

Many of these sections include sample code that demonstrates how to use the Movie Toolbox.

## Determining Whether the Movie Toolbox Is Installed

Use the Gestalt Manager to determine whether the Movie Toolbox is present. (The Gestalt Manager is fully described in *Inside Macintosh: Overview*.)

To determine whether the Movie Toolbox is available, use the Gestalt selector `gestaltQuickTime`. This selector has a value of `'qtim'`. If the Movie Toolbox is not installed, the Gestalt Manager returns an error.

For a description of how the version number is formatted, see the description of the numeric version part of the `'vers'` resource in the chapter “Gestalt Manager” in *Inside Macintosh: Overview*.

## Movie Toolbox

The code in Listing 2-1 contains a function that demonstrates how your application can call the Gestalt Manager.

---

**Listing 2-1** Using the Gestalt Manager with the Movie Toolbox

```
#include <GestaltEqu.h>
#include <Movies.h>

Boolean IsQuickTimeInstalled (void)
{
    short error;
    long result;

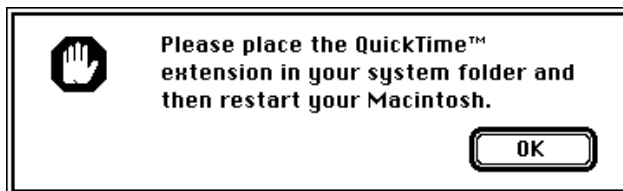
    error = Gestalt (gestaltQuickTime, &result);
    return (error == noErr);
}

void main (void)
{
    Boolean qtInstalled;
    .
    .
    .
    qtInstalled = IsQuickTimeInstalled ();
}
```

If you store movies inside your application document rather than just dealing with movie files, you must account for the possibility that a user's computer does not have QuickTime installed. If the Movie Toolbox is not available on a computer, your application can display a still-image representation of a movie in place of the movie itself. For example, you can store a PICT image from the movie in the document file, in addition to the movie itself. Your application can then display that image whenever the Movie Toolbox is unavailable. If the user tries to play the movie, you should inform the user that your application cannot play the movie by displaying an alert box like the one shown in Figure 2-25.

---

**Figure 2-25** An alert box that tells the user that QuickTime is unavailable



## Getting Ready to Work With Movies

The Movie Toolbox maintains state information for every application using it. In order to set up this information for your application, you must initialize the Movie Toolbox. You initialize the Movie Toolbox by calling the `EnterMovies` function (described on page 2-82).

You should call the `EnterMovies` function after you have initialized other Macintosh managers. Before calling this function you should make sure that the Movie Toolbox is available by calling the Gestalt Manager, as discussed in “Determining Whether the Movie Toolbox Is Installed” on page 2-33.

If you are writing a standard application, you do not need to call the `ExitMovies` function. Call the `ExitToShell` routine instead.

If you are writing a code resource, you may need to call the `ExitMovies` function (described on page 2-83), which allows the Movie Toolbox to clean up after your application has finished. After calling `ExitMovies`, you cannot make further calls to the Movie Toolbox.

## Getting a Movie From a File

Before your application can work with a movie, you must load the movie from its file. Your application must open the movie file and create a new movie from the movie stored in the file. You can then work with the movie. Use the `OpenMovieFile` function (described on page 2-98) to open a movie file. Use the `NewMovieFromFile` function (described on page 2-88) to load a movie from a movie file. The code in Listing 2-2 shows how you can use these functions.

**Listing 2-2** Getting a movie from a file

```
Movie GetMovie (void)
{
    OSErr          err;
    SFTypelist      typeList = {MovieFileType,0,0,0};
    StandardFileReply reply;
    Movie           aMovie = nil;
    short           movieResFile;

    StandardGetFilePreview (nil, 1, typeList, &reply);
    if (reply.sfGood)
    {
        err = OpenMovieFile (&reply.sfFile, &movieResFile,
                             fsRdPerm);

        if (err == noErr)
        {
            short      movieResID = 0;    /* want first movie */

```

## Movie Toolbox

```

    Str255      movieName;
    Boolean     wasChanged;

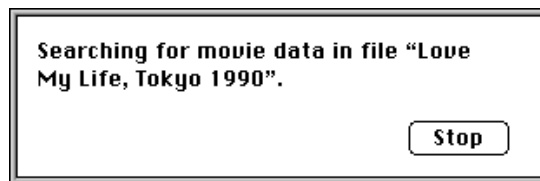
    err = NewMovieFromFile (&aMovie, movieResFile,
                           &movieResID,
                           movieName,
                           newMovieActive, /* flags */
                           &wasChanged);
    CloseMovieFile (movieResFile);
}
}
return aMovie;
}

```

QuickTime movies are stored in movie files. The Movie Toolbox uses the features of the Alias Manager and the new File Manager functions to manage a movie's references to its data (see "The Movie Toolbox and System 6" which begins on page 2-63 for more information about these features). A movie file does not necessarily contain the movie's data. The movie's data may reside in other files, which are referred to by the movie file.

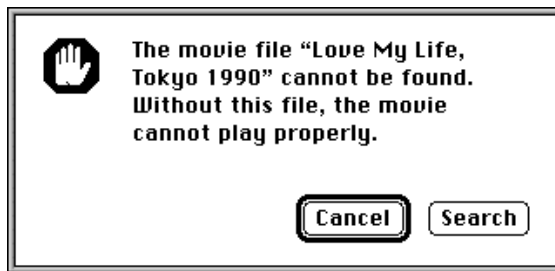
When your application instructs the Movie Toolbox to play a movie, the toolbox attempts to collect the movie's data. If the movie has become separated from its data, the Movie Toolbox uses the features of the Alias Manager to locate the data files. During this search, the Movie Toolbox automatically displays a dialog box similar to that shown in Figure 2-26. The user can cancel the search by clicking the Stop button.

**Figure 2-26** A dialog box used when searching for a movie's data



The Movie Toolbox performs a number of tests to verify that the file selected by the user is appropriate for the current movie. These tests include checking the creation date of the found file against the expected date and checking the size of the found file. The Movie Toolbox displays a dialog box similar to the one shown in Figure 2-27.

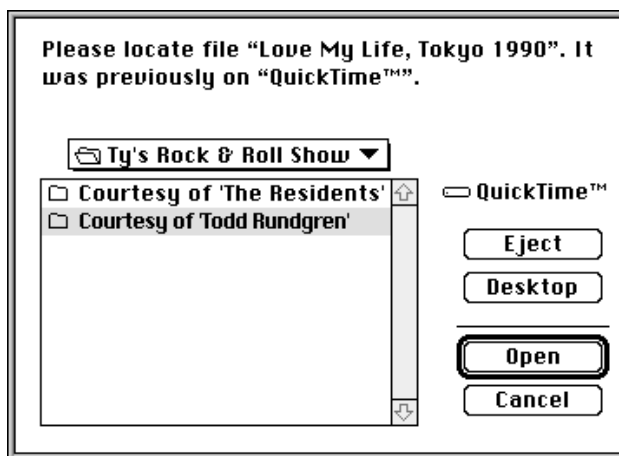


**Figure 2-27** A dialog box that informs the user the movie file cannot be found

The user has two options:

- by clicking Search, the user acknowledges the warning; the Movie Toolbox allows the user to locate a different data file
- by clicking Cancel, the user instructs the Movie Toolbox to ignore the current data reference—the Movie Toolbox tries to play the movie without the corresponding movie data

If the Movie Toolbox cannot locate a needed file, it displays a dialog box that allows the user to specify a file to try. Figure 2-28 shows a sample dialog box.

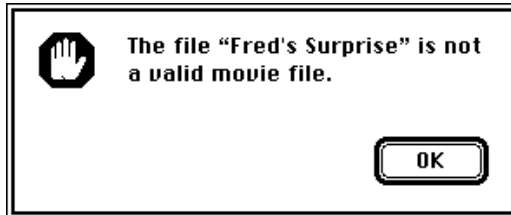
**Figure 2-28** A dialog box that allows the user to specify a movie file to try

## Movie Toolbox

If the user chooses a file that is not a valid movie file, it displays an alert similar to the one shown in Figure 2-29.

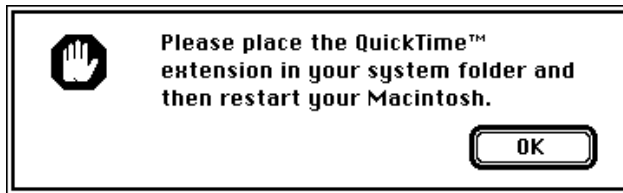
---

**Figure 2-29** An alert for an invalid movie file



---

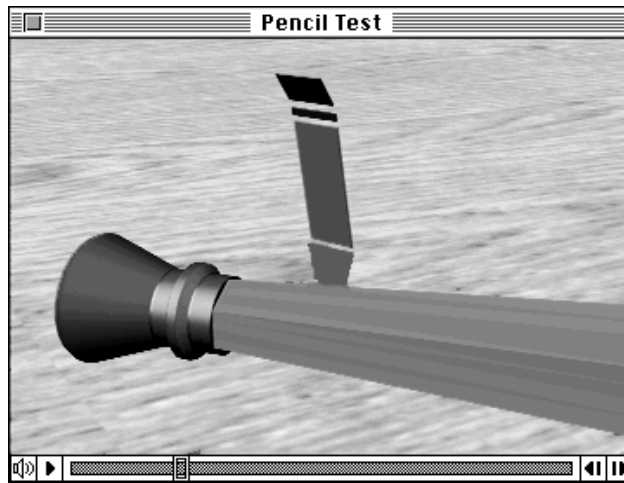
**Figure 2-30** An alert when QuickTime cannot be found



---

## Playing Movies With a Movie Controller

Movie controller components provide a simple method for displaying movies along with associated play controls. Using a movie controller component is the easiest way to incorporate a good movie player interface without having to write a substantial amount of code. A typical movie controller component allows the user to play a movie, make the movie pause, move forward and backward, and resize the movie's display. Some movie controllers may allow the user to edit the movie as well. Figure 2-31 shows Apple's movie controller.

**Figure 2-31** A movie controller playing a movie

Listing 2-3 shows how to play a movie using a movie controller component. This program uses the `GetMovie` function that is defined in Listing 2-2 on page 2-35. Refer to *Inside Macintosh: QuickTime Components* for a complete description of movie controller components and how to use them.

**Listing 2-3** Playing a movie using a movie controller component

```
#include <Types.h>
#include <Memory.h>
#include <Traps.h>
#include <Menus.h>
#include <Fonts.h>
#include <Packages.h>
#include <GestaltEqu.h>
#include <StandardFile.h>
#include <QDOffscreen.h>
#include "Movies.h"
#include "ImageCompression.h"
#include "QuickTimeComponents.h"

void main (void)
{
```

## Movie Toolbox

```

Movie Controller  aController;
WindowPtr        aWindow;
Rect             aRect;
Movie            aMovie;
Boolean          done = false;
OSErr            err;
EventRecord      theEvent;
WindowPtr        whichWindow;
short            part;

InitGraf (&qd.thePort);
InitFonts ();
InitWindows ();
InitMenus ();
TEInit ();
InitDialogs (nil);
err = EnterMovies ();
i

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

SetPort (aWindow);
aMovie = GetMovie ();
if (aMovie == nil) return;

SetRect(&aRect, 0, 0, 100, 100);
aController = NewMovieController (aMovie, &aRect,
                                 mcTopLeftMovie);

if (aController == nil) return;

err = MCGetControllerBoundsRect(aController, &aRect);
SizeWindow (aWindow, aRect.right,
           aRect.bottom, true);
ShowWindow (aWindow);
err = MCDoAction (aController,
                 mcActionSetKeysEnabled, (Ptr) true);

while (!done)
{
    WaitNextEvent(everyEvent, &theEvent, 0, nil );
    if (!MCIsPlayerEvent(aController, &theEvent))
    {
        switch (theEvent.what)

```

```

    {
        case updateEvt:
            whichWindow = (WindowPtr)theEvent.message;
            BeginUpdate (whichWindow);
            EraseRect (&whichWindow->portRect);
            EndUpdate (whichWindow);
            break;
        case mouseDown:
            part = FindWindow (theEvent.where,
                              &whichWindow);
            if (whichWindow == aWindow)
            {
                switch (part)
                {
                    case inGoAway:
                        done = TrackGoAway (whichWindow,
                                              theEvent.where);
                        break;

                    case inDrag:
                        DragWindow (whichWindow,
                                    theEvent.where,
                                    &qd.screenBits.bounds);
                        break;
                }
            }
        }
    }
}
DisposeMovieController (aController);
DisposeMovie (aMovie);
DisposeWindow(aWindow);
}

```

## Playing a Movie

The easiest way to play a movie is to use a movie controller component. See the previous section for more information about using movie controller components. If you want to create your own control for playing movies, you should observe the following guidelines:

- Your application should allow the user to manipulate movies in the same way that your application allows the user to work with static graphics—the user should be able to select, resize, cut, copy, and paste movies.
- Your application should save the current position of each movie in a document.

## Movie Toolbox

- Your application should not automatically play the movies in a document when the user opens the document.
- You should keep your movie controls simple and close to the movie.
- You should be consistent in the way that you allow the user to play a movie. Do not use single-clicking and double-clicking for the same thing. In general, use a single click to select a movie and use a double click to play it.
- When printing, your application should print each movie's current frame. You may choose to allow the user to select the frame for each movie, perhaps by means of a special menu item. Be sure not to print any of the user controls.

Once you have loaded a movie, you can play the movie. Your application must perform the following tasks:

1. Create a window for the movie to play in.
2. Position the movie in the window.
3. Start the movie.
4. Play the movie until it is done.
5. Dispose of the movie when it is done playing.

When you play a movie, the Movie Toolbox processes the movie's data in the context of the movie's time coordinate system. If the movie contains video data, the Movie Toolbox displays the resulting image in the display window you specify. If the movie contains audio data, the Movie Toolbox plays that sound track at the volume you set.

You must call the `MoviesTask` function (described on page 2-124) repeatedly until the movie is done playing. Each time you call the `MoviesTask` function, the Movie Toolbox processes the movie you are playing, updates the display as appropriate, and uses the Sound Manager to play the movie's sound. You can use the `IsMovieDone` function (described on page 2-125) to determine when the movie is finished playing.

The code in Listing 2-4 shows the steps your application must follow in order to play a movie. This program retrieves a movie, sizes the window properly, plays the movie forward, and exits. This program uses the `GetMovie` function, shown in Listing 2-2 on page 2-35 to retrieve a movie from a movie file. The movie controller component supplied by Apple also plays a movie. For more information, see the chapter "Movie Controller Components" in *Inside Macintosh: QuickTime Components*.

---

**Listing 2-4**      Playing a movie

```
#include <Types.h>
#include <Traps.h>
#include <Menus.h>
#include <Fonts.h>
#include <Packages.h>
```

## Movie Toolbox

```

#include <GestaltEqu.h>
#include "Movies.h"
#include "ImageCompression.h"

/* #include "QuickTimeComponents.h" */

#define doTheRightThing 5000

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

    InitGraf (&qd.thePort);
    InitFonts ();
    InitWindows ();
    InitMenus ();
    TEInit ();
    InitDialogs (nil);
    err = EnterMovies ();
    if (err) return;

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

    SetPort (aWindow);
    aMovie = GetMovie ();
    if (aMovie == nil) return;

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

    SizeWindow (aWindow, movieBox.right, movieBox.bottom, true);
    ShowWindow (aWindow);

```

## Movie Toolbox

```

SetMovieGWorld (aMovie, (CGrafPtr)aWindow, nil);

StartMovie (aMovie);

while ( !IsMovieDone(aMovie) && !done )
{
    if (WaitNextEvent (everyEvent, &theEvent, 0, nil))
    {
        switch ( theEvent.what )
        {
            case updateEvt:
                whichWindow = (WindowPtr)theEvent.message;
                if (whichWindow == aWindow)
                {
                    BeginUpdate (whichWindow);
                    UpdateMovie(aMovie);
                    SetPort (whichWindow);
                    EraseRect (&whichWindow->portRect);
                    EndUpdate (whichWindow);
                }
                break;

            case mouseDown:
                part = FindWindow (theEvent.where,
                                   &whichWindow);
                if (whichWindow == aWindow)
                {
                    switch (part)
                    {
                        case inGoAway:
                            done = TrackGoAway (whichWindow,
                                                  theEvent.where);

                            break;

                        case inDrag:
                            DragWindow (whichWindow,
                                         theEvent.where,
                                         &qd.screenBits.bounds);

                            break;

                    }
                }
                break;
        }
    }
}

```



## Movie Toolbox

```

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

```

## Movies and the Scrap

The Movie Toolbox makes it very easy for your application to deal with the scrap by providing two high-level functions that handle the details for you. When you want to put a movie onto the scrap, call the `PutMovieOnScrap` function (described on page 2-244). When you want to get a movie from the scrap, use the `NewMovieFromScrap` function (described on page 2-245).

When you use these functions, the Movie Toolbox takes care of all of the appropriate resources. For example, when you call the `PutMovieOnScrap` function, the Movie Toolbox creates a movie resource and a PICT image from the movie, and it places both on the scrap. In the future, as QuickTime grows, Apple will maintain these functions so that they continue to handle the appropriate resources.

## Creating a Movie

Creating a movie involves several steps. You must first create and open the movie file that is to contain the movie. You then create the tracks and media structures for the movie. You then add samples to the media structures. Finally, you add the movie resource to the movie file. The sample program in this section, `CreateWayCoolMovie`, demonstrates this process.

This program has been divided into several segments. The main segment, `CreateMyCoolMovie`, creates and opens the movie file, then invokes other functions to create the movie itself. Once the data has been added to the movie, this function saves the movie in its movie file and closes the file.

The `CreateMyCoolMovie` function uses the `CreateMyVideoTrack` and `CreateMySoundTrack` functions to create the movie's tracks. The `CreateMyVideoTrack` function creates the video track and the media that contains the track's data. It then collects sample data in the media by calling the `AddVideoSamplesToMedia` function. Note that this function uses the Image Compression Manager. The `CreateMySoundTrack` function creates the sound track and the media that contains the sound. It then collects sample data by calling the `AddSoundSamplesToMedia` function.

### Note

Throughout this volume, *sound track* refers to a QuickTime movie track that contains sound—as opposed to a *soundtrack*, which denotes the entire audio presentation of a movie as filmgoers know it. Consequently, a soundtrack may be made up of one or more QuickTime sound tracks. ♦

## Movie Toolbox

## A Sample Program for Creating a Movie

---

The `CreateWayCoolMovie` program consists of a number of segments, many of which are not included in this sample. Omitted segments deal with general initialization logic and other common aspects of Macintosh programming. The `HandleEditMenu` function, shown in Listing 2-5, has been included here to show how to initialize the Movie Toolbox with the `EnterMovies` function.

---

**Listing 2-5**      Creating a movie: The main program

```
#include <Types.h>
#include <Traps.h>
#include <Menus.h>
#include <Packages.h>
#include <Memory.h>
#include <Errors.h>
#include <Fonts.h>

#include <QuickDraw.h>
#include <Resources.h>
#include <GestaltEqu.h>
#include <FixMath.h>
#include <Sound.h>
#include <string.h>

#include "Movies.h"
#include "ImageCompression.h"

void CheckError(OSErr error, Str255 displayString)
{
    if (error == noErr) return;
    if (displayString[0] > 0)
        DebugStr(displayString);
    ExitToShell();
}

void InitMovieToolbox (void)
{
    OSErr err;

    InitGraf (&qd.thePort);
    InitFonts ();
    InitWindows ();
```

## Movie Toolbox

```

        InitMenus ();
        TEInit ();
        InitDialogs (nil);
        err = EnterMovies ();
        CheckError (err, "\pEnterMovies" );
    }

void main( void )
{
    InitMovieToolbox ();
    CreateMyCoolMovie ();
}

```

### A Sample Function for Creating and Opening a Movie File

The `CreateMyCoolMovie` function, shown in Listing 2-6, contains the main logic for this program. This function creates and opens a movie file for the new movie. It then establishes a data reference for the movie's data (note that, if your movie's data is stored in the same file as the movie itself, you do not have to create a data reference—set the data reference to 0). This function then calls two other functions, `CreateMyVideoTrack` and `CreateMySoundTrack`, to create the tracks for the new movie. Once the tracks have been created, `CreateMyCoolMovie` adds the new resource to the movie file and closes the movie file.

**Listing 2-6** Creating and opening a movie file

```

#define kMyCreatorType 'TVOD'

/*
Sample Player's creator type since it is the movie player
of choice. You can use your own creator type, of course.
*/

#define kPrompt "\pEnter movie file name:"

void CreateMyCoolMovie (void)
{
    Point    where = {100,100};
    SFReply  theSFReply;
    Movie    theMovie = nil;
    FSSpec    mySpec;
    short    resRefNum = 0;
    short    resId = 0;
    OSErr    err = noErr;

```

## Movie Toolbox

```

SFPutFile (where, "\pEnter movie file name:",
           "\pMovie File", nil, &theSFReply);
if (!theSFReply.good) return;

FSMakeFSSpec(theSFReply.vRefNum, 0,
             theSFReply.fName, &mySpec);

err = CreateMovieFile (&mySpec,
                      'TVOD',
                      smCurrentScript,
                      createMovieFileDeleteCurFile,
                      &resRefNum,
                      &theMovie );
CheckError(err, "\pCreateMovieFile");

CreateMyVideoTrack (theMovie);
CreateMySoundTrack (theMovie);

err = AddMovieResource (theMovie, resRefNum, &resId,
                       theSFReply.fName);
CheckError(err, "\pAddMovieResource");

if (resRefNum) CloseMovieFile (resRefNum);
DisposeMovie (theMovie);
}

```

### A Sample Function for Creating a Video Track in a New Movie

---

The `CreateMyVideoTrack` function, shown in Listing 2-7, creates a video track in the new movie. This function creates the track and its media by calling the `NewMovieTrack` and `NewTrackMedia` functions, respectively. This function then establishes a media-editing session and adds the movie's data to the media. The bulk of this work is done by the `AddVideoSamplesToMedia` subroutine. Once the data has been added to the media, this function adds the media to the track by calling the Movie Toolbox's `InsertMediaIntoTrack` function (described on page 2-265).

**Listing 2-7** Creating a video track

```

#define kVideoTimeScale 600
#define kTrackStart      0
#define kMediaStart      0
#define kFix1             0x00010000

void CreateMyVideoTrack (Movie theMovie)
{
    Track      theTrack;
    Media      theMedia;
    OSErr      err = noErr;
    Rect       trackFrame = {0,0,100,320};

    theTrack = NewMovieTrack (theMovie,
                             FixRatio(trackFrame.right,1),
                             FixRatio(trackFrame.bottom,1),
                             kNoVolume);
    CheckError( GetMoviesError(), "\pNewMovieTrack" );

    theMedia = NewTrackMedia (theTrack, VideoMediaType,
                             600, // Video Time Scale
                             nil, 0);
    CheckError( GetMoviesError(), "\pNewTrackMedia" );

    err = BeginMediaEdits (theMedia);
    CheckError( err, "\pBeginMediaEdits" );

    AddVideoSamplesToMedia (theMedia, &trackFrame);

    err = EndMediaEdits (theMedia);
    CheckError( err, "\pEndMediaEdits" );

    err = InsertMediaIntoTrack (theTrack, 0, /* track start time */
                               0,          /* media start time */
                               GetMediaDuration (theMedia),
                               kFix1);
    CheckError( err, "\pInsertMediaIntoTrack" );
}

```

## A Sample Function for Adding Video Samples to a Media

---

The `AddVideoSamplesToMedia` function, shown in Listing 2-8, creates video data frames, compresses each frame, and adds the frames to the media. This function creates its own video data by calling the `DrawAFrame` function. Note that this function does not temporally compress the image sequence; rather, the function only spatially compresses each frame individually.

---

**Listing 2-8**      Adding video samples to a media

```
#define kSampleDuration      240
        /* video frames last 240 * 1/600th of a second */
#define kNumVideoFrames     29
#define kNoOffset           0
#define kMgrChoose          0
#define kSyncSample         0
#define kAddOneVideoSample  1
#define kPixelDepth         16

void AddVideoSamplesToMedia (Media theMedia,
                           const Rect *trackFrame)
{
    long                maxCompressedSize;
    GWorldPtr           theGWorld = nil;
    long                curSample;
    Handle               compressedData = nil;
    Ptr                 compressedDataPtr;
    ImageDescriptionHandle imageDesc = nil;
    CGrafPtr            oldPort;
    GDHandle             oldGDeviceH;
    OSErr               err = noErr;

    err = NewGWorld (&theGWorld,
                    16,          /* pixel depth */
                    trackFrame,
                    nil,
                    nil,
                    (GWorldFlags) 0 );
    CheckError (err, "\pNewGWorld");

    LockPixels (theGWorld->portPixMap);
```

## Movie Toolbox

```

err = GetMaxCompressionSize (theGWorld->portPixMap,
                             trackFrame,
                             0, /* let ICM choose depth */
                             codecNormalQuality,
                             'rle ',
                             (CompressorComponent) anyCodec,
                             &maxCompressedSize);
CheckError (err, "\pGetMaxCompressionSize" );

compressedData = NewHandle(maxCompressedSize);
CheckError( MemError(), "\pNewHandle" );

MoveHHi( compressedData );
HLock( compressedData );
compressedDataPtr = StripAddress( *compressedData );

imageDesc = (ImageDescriptionHandle)NewHandle(4);
CheckError( MemError(), "\pNewHandle" );

GetGWorld (&oldPort, &oldGDeviceH);
SetGWorld (theGWorld, nil);

for (curSample = 1; curSample < 30; curSample++)
{
    EraseRect (trackFrame);
    DrawFrame(trackFrame, curSample);

    err = CompressImage (theGWorld->portPixMap,
                         trackFrame,
                         codecNormalQuality,
                         'rle ',
                         imageDesc,
                         compressedDataPtr );
    CheckError( err, "\pCompressImage" );

    err = AddMediaSample(theMedia,
                        compressedData,
                        0, /* no offset in data */
                        (**imageDesc).dataSize,
                        60, /* frame duration = 1/10 sec */
                        (SampleDescriptionHandle)imageDesc,
                        1, /* one sample */

```

## Movie Toolbox

```

                                0,      /* self-contained samples */
                                nil);
    CheckError( err, "\pAddMediaSample" );
}

SetGWorld (oldPort, oldGDeviceH);

if (imageDesc) DisposeHandle ((Handle)imageDesc);
if (compressedData) DisposeHandle (compressedData);
if (theGWorld) DisposeGWorld (theGWorld);
}

```

### A Sample Function for Creating Video Data for a Movie

---

The `DrawAFrame` function, shown in Listing 2-9, creates video data for this movie. This function draws a different frame each time it is invoked, based on the sample number, which is passed as a parameter.

---

**Listing 2-9**     Creating video data

```

void DrawFrame (const Rect *trackFrame,  long curSample)
{
    Str255 numStr;

    ForeColor( redColor );
    PaintRect( trackFrame );

    ForeColor( blueColor );
    NumToString (curSample, numStr);
    MoveTo ( trackFrame->right / 2, trackFrame->bottom / 2);
    TextSize ( trackFrame->bottom / 3);
    DrawString (numStr);
}

```

### A Sample Function for Creating a Sound Track

---

The `CreateMySoundTrack` function, shown in Listing 2-10, creates the movie's sound track. This sound track is not synchronized to the video frames of the movie—rather, it is just a separate sound track that accompanies the video data. This function relies upon an 'snd' resource for its source sound. The `CreateMySoundTrack` function uses the `CreateSoundDescription` function to create the sound description structure for these samples.

As with the `CreateMyVideoTrack` function discussed earlier, this function creates the track and its media by calling the `NewMovieTrack` and `NewTrackMedia` functions,



respectively. This function then establishes a media-editing session and adds the movie's data to the media. This function adds the sound samples using a single invocation of the `AddMediaSample` function. This is possible because all the sound samples are the same size and rely on the same sample description (the `SoundDescription` structure). If you use this approach, it is often advisable to break up the sound data in the movie, so that the movie plays smoothly. After you create the movie, you can call the `FlattenMovie` function (described on page 2-105) to create an interleaved version of the movie. Another approach is to call `AddMediaSample` multiple times, breaking the sound into multiple chunks at that point.

Once the data has been added to the media, this function adds the media to the track by calling the Movie Toolbox's `InsertMediaIntoTrack` function (described on page 2-265).

---

**Listing 2-10**     Creating a sound track

```
#define kSoundSampleDuration 1
#define kSyncSample 0
#define kTrackStart 0
#define kMediaStart 0
#define kFixl          0x00010000

void CreateMySoundTrack (Movie theMovie)
{
    Track          theTrack;
    Media          theMedia;
    Handle         sndHandle = nil;
    SoundDescriptionHandle sndDesc = nil;
    long           sndDataOffset;
    long           sndDataSize;
    long           numSamples;
    OSErr          err = noErr;

    sndHandle = GetResource ('snd ', 128);
    CheckError (ResError(), "\pGetResource" );
    if (sndHandle == nil) return;

    sndDesc = (SoundDescriptionHandle) NewHandle(4);
    CheckError (MemError(), "\pNewHandle" );

    CreateSoundDescription (sndHandle,
                           sndDesc,
```

## Movie Toolbox

```

        &sndDataOffset,
        &numSamples,
        &sndDataSize );

theTrack = NewMovieTrack (theMovie, 0, 0, kFullVolume);
CheckError (GetMoviesError(), "\pNewMovieTrack" );

theMedia = NewTrackMedia (theTrack, SoundMediaType,
                          FixRound ((*sndDesc).sampleRate),
                          nil, 0);
CheckError (GetMoviesError(), "\pNewTrackMedia" );

err = BeginMediaEdits (theMedia);
CheckError( err, "\pBeginMediaEdits" );

err = AddMediaSample(theMedia,
                    sndHandle,
                    sndDataOffset, /* offset in data */
                    sndDataSize,
                    1,              /* duration of each sound sample */
                    (SampleDescriptionHandle) sndDesc,
                    numSamples,
                    0,              /* self-contained samples */
                    nil);
CheckError( err, "\pAddMediaSample" );

err = EndMediaEdits (theMedia);
CheckError( err, "\pEndMediaEdits" );

err = InsertMediaIntoTrack (theTrack,
                           0,      /* track start time */
                           0,      /* media start time */
                           GetMediaDuration (theMedia),
                           kFix1);
CheckError( err, "\pInsertMediaIntoTrack" );

if (sndDesc != nil) DisposeHandle( (Handle)sndDesc);
}

```

## A Sample Function for Creating a Sound Description Structure

The `CreateSoundDescription` function, shown in Listing 2-11, creates a sound description structure that correctly describes the sound samples obtained from the 'snd' resource. This function can handle all the sound data formats that are possible in the sound resource. This function uses the `GetSndHdrOffset` function to locate the sound data in the sound resource.

**Listing 2-11** Creating a sound description

```

/* Constant definitions */
/*
    for the following constants, please consult the Macintosh
    Audio Compression and Expansion Toolkit
*/
#define kMACEBeginningNumberOfBytes 6
#define kMACE31MonoPacketSize 2
#define kMACE31StereoPacketSize 4
#define kMACE61MonoPacketSize 1
#define kMACE61StereoPacketSize 2

void CreateSoundDescription (Handle sndHandle,
                            SoundDescriptionHandle sndDesc,
                            long *sndDataOffset,
                            long *numSamples,
                            long *sndDataSize )
{
    long                sndHdrOffset = 0;
    long                sampleDataOffset;
    SoundHeaderPtr      sndHdrPtr = nil;
    long                numFrames;
    long                samplesPerFrame;
    long                bytesPerFrame;
    SignedByte          sndHState;
    SoundDescriptionPtr sndDescPtr;

    *sndDataOffset = 0;
    *numSamples = 0;
    *sndDataSize = 0;

    SetHandleSize( (Handle)sndDesc, sizeof(SoundDescription) );
    CheckError(MemError(), "\pSetHandleSize");
}

```

## Movie Toolbox

```

sndHdrOffset = GetSndHdrOffset (sndHandle);
if (sndHdrOffset == 0) CheckError(-1,  "\\pGetSndHdrOffset ");

    /* we can use pointers since we don't move memory */
sndHdrPtr = (SoundHeaderPtr) (*sndHandle + sndHdrOffset);
sndDescPtr = *sndDesc;

sndDescPtr->descSize = sizeof (SoundDescription);
    /* total size of sound description structure */
sndDescPtr->resvd1 = 0;
sndDescPtr->resvd2 = 0;
sndDescPtr->dataRefIndex = 1;
sndDescPtr->compressionID = 0;
sndDescPtr->packetSize = 0;
sndDescPtr->version = 0;
sndDescPtr->revlevel = 0;
sndDescPtr->vendor = 0;

switch (sndHdrPtr->encode)
{
    case stdSH:
        sndDescPtr->dataFormat = 'raw ';
        /* uncompressed offset-binary data */
        sndDescPtr->numChannels = 1;
        /* number of channels of sound */
        sndDescPtr->sampleSize = 8;
        /* number of bits per sample */
        sndDescPtr->sampleRate = sndHdrPtr->sampleRate;
        /* sample rate */
        *numSamples      = sndHdrPtr->length;
        *sndDataSize     = *numSamples;
        bytesPerFrame   = 1;
        samplesPerFrame = 1;
        sampleDataOffset = (Ptr)&sndHdrPtr->sampleArea
                           - (Ptr)sndHdrPtr;

        break;

    case extSH:
        {
            ExtSoundHeaderPtr    extSndHdrP;

            extSndHdrP = (ExtSoundHeaderPtr)sndHdrPtr;

```

```

    sndDescPtr->dataFormat = 'raw ';
        /* uncompressed offset-binary data */
    sndDescPtr->numChannels = extSndHdrP->numChannels;
        /* number of channels of sound */
    sndDescPtr->sampleSize = extSndHdrP->sampleSize;
        /* number of bits per sample */
    sndDescPtr->sampleRate = extSndHdrP->sampleRate;
        /* sample rate */
    numFrames = extSndHdrP->numFrames;
    *numSamples = numFrames;
    bytesPerFrame = extSndHdrP->numChannels *
        ( extSndHdrP->sampleSize / 8);
    samplesPerFrame = 1;
    *sndDataSize = numFrames * bytesPerFrame;
    sampleDataOffset = (Ptr>(&extSndHdrP->sampleArea)
        - (Ptr)extSndHdrP;
}
break;

case cmpSH:
{
    CmpSoundHeaderPtr cmpSndHdrP;

    cmpSndHdrP = (CmpSoundHeaderPtr)sndHdrPtr;
    sndDescPtr->numChannels = cmpSndHdrP->numChannels;
        /* number of channels of sound */
    sndDescPtr->sampleSize = cmpSndHdrP->sampleSize;
        /* number of bits per sample before compression */
    sndDescPtr->sampleRate = cmpSndHdrP->sampleRate;
        /* sample rate */
    numFrames = cmpSndHdrP->numFrames;
    sampleDataOffset = (Ptr>(&cmpSndHdrP->sampleArea)
        - (Ptr)cmpSndHdrP;
    switch (cmpSndHdrP->compressionID)
    {
        case threeToOne:
            sndDescPtr->dataFormat = 'MAC3';
            /* compressed 3:1 data */
            samplesPerFrame = kMACEBeginningNumberOfBytes;
            *numSamples = numFrames * samplesPerFrame;
            switch (cmpSndHdrP->numChannels)
            {
                case 1:

```

```

        bytesPerFrame = cmpSndHdrP->numChannels
                        * kMACE31MonoPacketSize;
        break;
    case 2:
        bytesPerFrame = cmpSndHdrP->numChannels
                        * kMACE31StereoPacketSize;
        break;
    default:
        CheckError(-1, "\pCorrupt sound data" );
        break;
    }
    *sndDataSize = numFrames * bytesPerFrame;
    break;
case sixToOne:
    sndDescPtr->dataFormat = 'MAC6';
    /* compressed 6:1 data */
    samplesPerFrame = kMACEBeginningNumberOfBytes;
    *numSamples = numFrames * samplesPerFrame;
    switch (cmpSndHdrP->numChannels)
    {
        case 1:
            bytesPerFrame = cmpSndHdrP->numChannels
                            * kMACE61MonoPacketSize;
            break;
        case 2:
            bytesPerFrame = cmpSndHdrP->numChannels
                            * kMACE61StereoPacketSize;
            break;
        default:
            CheckError(-1, "\pCorrupt sound data" );
            break;
    }
    *sndDataSize = (*numSamples) * bytesPerFrame;
    break;
default:
    CheckError(-1, "\pCorrupt sound data" );
    break;
}
}
break; /* switch cmpSndHdrP->compressionID:*/
/* of cmpSH: */

default:
    CheckError(-1, "\pCorrupt sound data" );

```

```

        break;

    }

    /* switch sndHdrPtr->encode */
    *sndDataOffset = sndHdrOffset + sampleDataOffset;
}

```

## Parsing a Sound Resource

The `GetSndHdrOffset` function, shown in Listing 2-12, parses the specified sound resource and locates the sound data stored in the resource. The `GetSndHdrOffset` function cruises through a specified 'snd' resource. It locates the sound data, if any, and returns its type, offset, and size into the resource.

The `GetSndHdrOffset` function returns an offset instead of a pointer so that the data is not locked in memory. By returning an offset, the calling function can decide when and if it wants the resource locked down to access the sound data.

The first step in finding this data is to determine if the 'snd' resource is format (type) 1 or format (type) 2. A type 2 is easy, but a type 1 requires that you find the number of 'snth' resource types specified and then skip over each one, including the `init` option. Once you do this, you have a pointer to the number of commands in the 'snd' resource. When the function finds the first one, it examines the command to find out if it is a sound data command. Since it is a sound resource, the command also has its `dataPointerFlag` parameter set to 1. When the function finds a sound data command, it returns its offset and type, and exits.

### ▲ WARNING

Do not send the `GetSndHdrOffset` function a nil handle; if you do, your system will crash. ▲

**Listing 2-12** Parsing a sound resource

```

typedef SndCommand *SndCmdPtr;

typedef struct
{
    short    format;
    short    numSynths;
} Snd1Header, *Snd1HdrPtr, **Snd1HdrHndl;

typedef struct
{
    short    format;
    short    refCount;
} Snd2Header, *Snd2HdrPtr, **Snd2HdrHndl;

```

## Movie Toolbox

```

typedef struct
{
    short    synthID;
    long     initOption;
} SynthInfo, *SynthInfoPtr;

long GetSndHdrOffset (Handle sndHandle)
{
    short howManyCmds;
    long sndOffset = 0;
    Ptr sndPtr;

    if (sndHandle == nil) return 0;
    sndPtr = *sndHandle;
    if (sndPtr == nil) return 0;

    if ((* (Snd1HdrPtr)sndPtr).format == firstSoundFormat)
    {
        short synths = ((Snd1HdrPtr)sndPtr)->numSynths;
        sndPtr += sizeof(Snd1Header) + (sizeof(SynthInfo) * synths);
    } else
    {
        sndPtr += sizeof(Snd2Header);
    }

    howManyCmds = *(short *)sndPtr;

    sndPtr += sizeof(howManyCmds);
    /*
    sndPtr is now at the first sound command--cruise all
    commands and find the first soundCmd or bufferCmd
    */
    while (howManyCmds > 0)
    {
        switch (((SndCmdPtr)sndPtr)->cmd)
        {
            case (soundCmd + dataOffsetFlag):
            case (bufferCmd + dataOffsetFlag):
                sndOffset = ((SndCmdPtr)sndPtr)->param2;
                howManyCmds = 0; /* done, get out of loop */
                break;
            default:
                /* catch any other type of commands */

```



## Movie Toolbox

```

        sndPtr += sizeof(SndCommand);
        howManyCmds--;
        break;
    }
} /* done with all commands */

return sndOffset;
} /* of GetSndHdrOffset */

```

## Saving Movies in Movie Files

The Movie Toolbox allows you to save movies in movie files. Movie files have a file type of 'Moov'. Typically, the movie itself is stored in the resource fork of the movie file. The movie's data may reside in the data fork of the movie file, or in other files.

When you create a new movie, you must create a file to contain the movie data. Use the `CreateMovieFile` function (described on page 2-96) to create a new movie file. This function returns a file system reference number that you must use to identify the file to other Movie Toolbox functions. You can add your movie to the file by calling the `AddMovieResource` function (described on page 2-102). When you are done with the file, you close it by calling the `CloseMovieFile` function (described on page 2-99). Your movie is now safely stored in the movie file.

If you are working with an existing movie, you must read that movie from a movie file or choose a movie from the scrap. You first open the movie file by calling the `OpenMovieFile` function (described on page 2-98). You then load the movie from that file by calling the `NewMovieFromFile` function (described on page 2-88). Alternatively, you can use the `NewMovieFromHandle` function (described on page 2-90). After you have edited the movie, you must store it in your file if you want to save your changes. If you want to replace the old movie, use the `UpdateMovieResource` function (described on page 2-103). If you want to keep the old movie, create a new movie by calling the `AddMovieResource` function described on page 2-102 (a movie file may contain more than one movie resource). You should then close the movie file by calling the `CloseMovieFile` function (described on page 2-99).

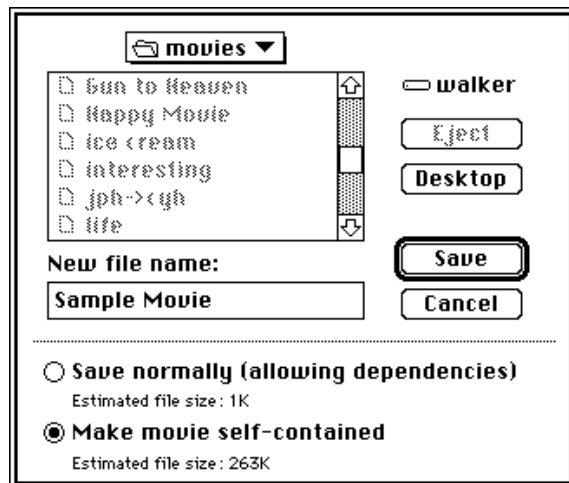
The Movie Toolbox maintains a changed flag for each movie your application loads. You can use this flag to determine when to save your movie. The Movie Toolbox sets this flag to `true` whenever you make a change to a movie that should be saved. You can read this flag by calling the `HasMovieChanged` function (described on page 2-101). You can set the flag to `false` by calling the `ClearMovieChanged` function (described on page 2-102).

The Movie Toolbox provides two functions for deleting movies: `DeleteMovieFile` and `RemoveMovieResource`. Use `DeleteMovieFile` (described on page 2-100) to delete a movie file. Use `RemoveMovieResource` (described on page 2-104) to delete a movie from a movie file. Don't use the corresponding standard Macintosh Toolbox routines (`FSpDelete` and `RmveResource`). The Movie Toolbox maintains movie references between files correctly whereas these routines do not.

## Movie Toolbox

The Movie Toolbox allows you to create movie files that contain all of their movie data, rather than containing references to data in other files. This may be necessary when creating a version of a movie that is to be moved to another computer system. The Movie Toolbox also accommodates operating systems that do not recognize files that contain more than one fork. In this case, you can use the `FlattenMovie` or `FlattenMovieData` functions (described on page 2-105 and page 2-107, respectively) to create a movie file that stores the movie and all of its data in the data fork of a Macintosh file. You can then transfer that file to another operating system. Your application may allow the user to decide how to save the movie. In this case, you can use a Save As dialog box similar to the one shown in Figure 2-32. In this dialog box, the user can elect to create a movie file that contains all of the data for a movie by clicking the “Make movie self-contained” radio button.

**Figure 2-32** A sample movie Save As dialog box



## Using Movies in Your Event Loop

Your application needs to grant time to the Movie Toolbox to allow your movies to play. To do this, you call the `MoviesTask` function from your main event loop. The `MoviesTask` function (described on page 2-124) instructs the Movie Toolbox to service all your active movies. Call `MoviesTask` regularly so that your movie can play smoothly. You can use the `UpdateMovie` function to force your movie to be redrawn after it has been uncovered. It will not be redrawn until the next call to `MoviesTask`.

Your application should call `UpdateMovie` between the Window Manager's `BeginUpdate` and `EndUpdate` functions. (For details on `BeginUpdate` and `EndUpdate`, see *Inside Macintosh: Macintosh Toolbox Essentials*.) Do not call `MoviesTask` at this time. You will observe better display behavior if you call `MoviesTask` at the end of your update processing.

## Movie Toolbox

The code shown in Listing 2-13 demonstrates the use of the `UpdateMovie` function in a Window Manager update sequence. For the Movie Toolbox to know that it has to display (or update) a movie when `MoviesTask` is called, you must call `UpdateMovie` as shown. If you are using the movie controller component and call the `MCIsPlayerEvent` function, you do not need to call `UpdateMovie` in response to an update event. (See the chapter “Movie Controller Component” in *Inside Macintosh: QuickTime Components*, for details on `MCIsPlayerEvent`.)

**Note**

Contrary to normal update handling, where applications draw to the window in between calls to `BeginUpdate` and `EndUpdate`, you should not call `MoviesTask`. ♦

The `UpdateMovie` function tells the Movie Toolbox that a portion of the movie has been invalidated. However, it is not redrawn until `MoviesTask` is called.

**Listing 2-13** Handling movie update events

```
#include <Events.h>
#include <ToolUtils.h>
#include "Movies.h"

void DoUpdate (WindowPtr theWindow, Movie theMovie)
{
    BeginUpdate (theWindow);
    UpdateMovie (theMovie);
    EndUpdate (theWindow);
} /* DoUpdate */
```

## The Movie Toolbox and System 6

The Movie Toolbox makes extensive use of some of the facilities of System 7. In particular, the toolbox uses the features of the Alias Manager and the new File Manager routines that support the `FSSpec` data type. In order to allow you to use QuickTime on Macintosh computers that are running System 6, QuickTime provides its own support for these features.

This section discusses the details of the Movie Toolbox’s support. For a complete description of the Alias Manager and File Manager features of System 7, refer to *Inside Macintosh: Files*.

**Note**

Track mattes are approximated. The System 7 version of the Time Manager is installed, but not its Gestalt selector. ♦

## The Alias Manager

---

When you run the Movie Toolbox on a Macintosh computer that is running System 6, QuickTime installs a limited version of the Alias Manager. This version of the Alias Manager supports most of the routines that are supported by the standard manager. In addition, aliases you create in System 6 are completely compatible with those you create in System 7. However, the limited version of the Alias Manager does not support relative aliases, does not search multiple volumes, does not support exhaustive searches, and does not mount network volumes.

The following list provides more detailed information about this limited version of the Alias Manager.

- The `NewAlias` function is supported and accepts a `fromFile` specification; however, the function does not create relative aliases.
- The `NewAliasMinimalFromFullPath` function is not supported.
- The `ResolveAlias` function is supported and accepts a `fromFile` specification; however, the function ignores this parameter.
- The `ResolveAliasFile` function is not supported.
- The `MatchAlias` function is supported, but it ignores the `kARMSearchMore`, `kARMSearchRelFirst`, and `kARMMultVols` options of the `rulesMask` parameter.
- The `UpdateAlias` function is supported and accepts a `fromFile` specification; however, the function ignores this parameter.

### Note

This limited version of the Alias Manager does not install the Alias Manager's Gestalt selector. If your application relies on more support than this version supplies, be sure to examine the Alias Manager's Gestalt selector. ♦

## The File Manager

---

The Movie Toolbox uses the File Manager functions that support the file system specification structures (of type `FSSpec`). When you use QuickTime on Macintosh computers that are running System 6, QuickTime installs support for most of the new File Manager routines. These routines behave the same as they do in System 7.

Specifically, QuickTime provides support for the following File Manager functions that use the `FSSpec` data type:

<code>FSMakeFSSpec</code>	<code>FSpOpenDF</code>
<code>FSpOpenRF</code>	<code>FSpCreate</code>
<code>FSpDirCreate</code>	<code>FSpDelete</code>
<code>FSpGetFInfo</code>	<code>FSpSetFInfo</code>
<code>FSpSetFLock</code>	<code>FSpRstFLock</code>

## Movie Toolbox

`FSpRename`                      `FSpCatMove`  
`FSpOpenResFile`            `FSpCreateResFile`  
`FSpGetCatInfo`

QuickTime does not support the `FSpExchangeFiles` function.

**Note**

QuickTime does not install the File Manager's Gestalt selector for the functions that support the `FSSpec` data type. If QuickTime is installed, you can assume that these File Manager functions are supported, even if `gestaltHasFSSpecCalls` is not set. ♦

## Previewing Files

---

QuickTime includes extensions to the Standard File Package that allow you to create and display file previews—information that gives the user an idea of a file's contents without opening the file. Typically, a file's preview is a small PICT image (called a *thumbnail*), but previews may also contain other types of information that is appropriate to the type of file being considered. For example, a text file's preview might tell the user when the file was created and what it discusses. You can use the Image Compression Manager to create thumbnail images—see the chapter “Image Compression Manager” later in this book for more information about thumbnail images.

QuickTime provides new standard file functions that your application can use to display a file's preview during the Open dialog box. These functions allow your application to support previews automatically.

**Note**

Before using these new standard file functions, make sure that the Image Compression Manager is installed. See the chapter “Image Compression Manager” in this book for information about the Image Compression Manager's Gestalt selector. ♦

In addition, the Movie Toolbox includes two functions that allow you to create a preview for a file.

## Previewing Files in System 6 Using Standard File Reply Structures

---

The Movie Toolbox provides two new standard file functions that allow you to display file previews in an Open dialog box in System 6 using standard file reply structures: `SFGetFilePreview` and `SFPGetFilePreview`. The `SFGetFilePreview` function (described on page 2-306) corresponds to the existing `SFGetFile` function; the `SFPGetFilePreview` function (described on page 2-308) corresponds to the existing `SFPGetFile` function. Both of these new functions take the same parameters as their existing counterparts. For information about `SFGetFile` and `SFPGetFile`, see *Inside Macintosh: Files*.

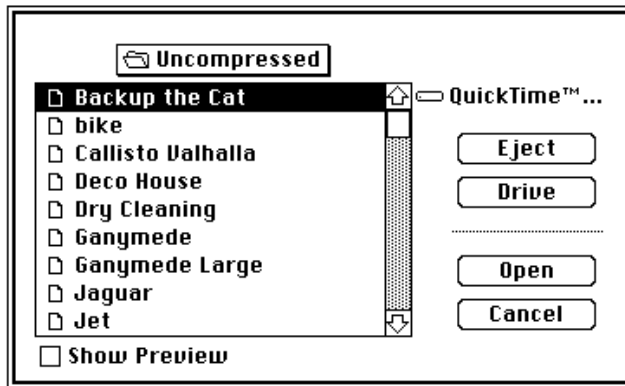
## Movie Toolbox

**IMPORTANT**

All the functions for previewing files are present in System 6 except the CustomGetFilePreview function. The StandardGetFilePreview function is preferable and will work on System 6. ▲

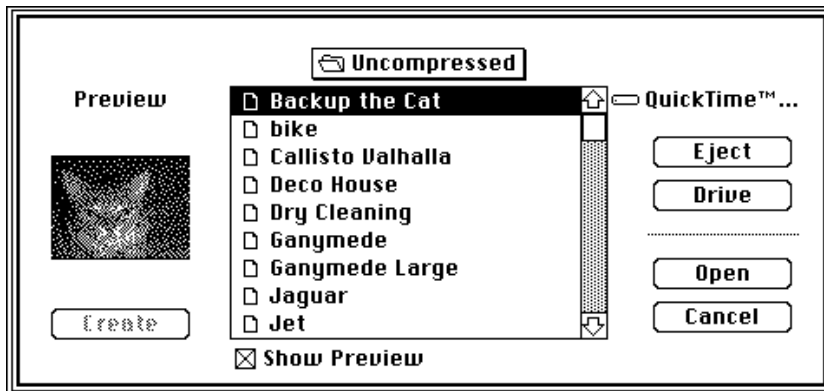
The SFGGetFilePreview function uses the dialog box shown in Figure 2-33. The SFPGetFilePreview function can also use this dialog box, if you do not supply your own.

**Figure 2-33** SFGGetFilePreview or SFPGetFilePreview dialog box without preview



You use these new functions in place of the existing standard file functions to indicate whether or not you want to allow the user to display previews during the Open dialog box. The user displays a file's preview by selecting a file in the dialog box and clicking Show Preview. When the user does so, the functions display the preview for the file, as shown in Figure 2-34.

**Figure 2-34** SFGGetFilePreview or SFPGetFilePreview dialog box with preview



The preview area of the dialog box is displayed whenever previewing is enabled.

## Customizing Your Interface in System 6

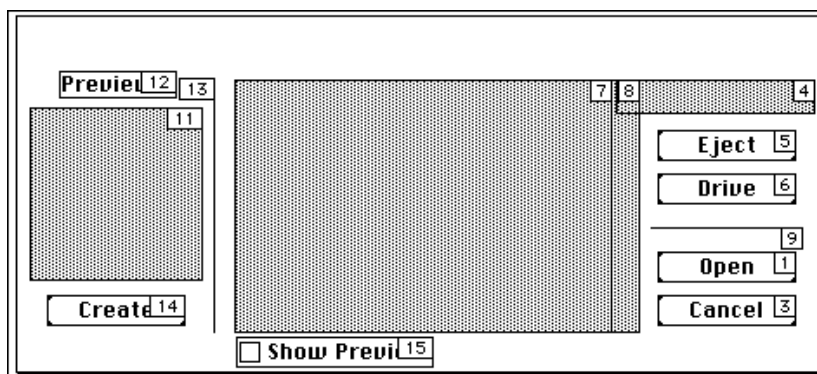
If your application requires it, you can customize the user interface for identifying files. The `SFGetFilePreview` function does not allow you to use a custom dialog box by creating your own dialog template resource. However, the `SFPGetFilePreview` function does let you access a custom dialog box of any resource type with the `dlgID` parameter.

Figure 2-35 shows the standard dialog box used by `SFPGetFilePreview` and `SFGetFilePreview`. Your dialog box and dialog filter function must support at least these dialog items.

### Note

Alter the dialog boxes only if necessary. Apple does not guarantee future compatibility if you use a customized dialog box. ♦

**Figure 2-35** Standard preview dialog box for `SFGetFilePreview` and `SFPGetFilePreview`



Items to the left of item 13 are visible only when previewing. If you want to define items that are visible only during a file preview, place them to the left of item 13 in your custom dialog box.

If your application defines a custom dialog box, be sure to include the following items in your dialog box definition:

```
enum
{
    /* dialog items to include in dialog box definition for use
       with SFGetFilePreview function
    */
    sfpItemPreviewAreaUser      = 11,    /* user preview area */
    sfpItemPreviewStaticText    = 12,    /* static text preview */

```

## Movie Toolbox

```

    sfpItemPreviewDividerUser = 13,      /* user divider preview */
    sfpItemCreatePreviewButton = 14,     /* create preview button */
    sfpItemShowPreviewButton  = 15      /* show preview button */
};

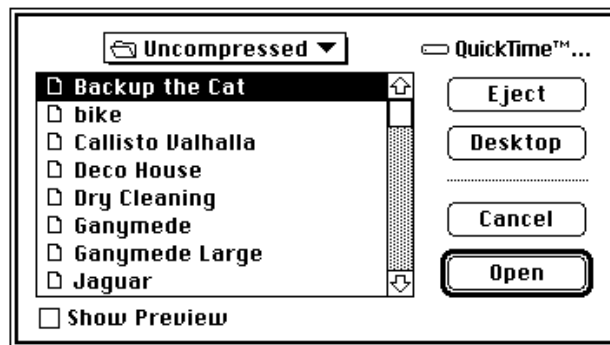
```

## Previewing Files in System 7 Using Standard File Reply Structures

The Movie Toolbox provides two new standard file functions, `standardGetFilePreview` and `CustomGetFilePreview`, that allow you to display file previews in an Open dialog box in System 7 using standard file reply structures (of type `StandardFileReply`). The `StandardGetFilePreview` function (described on page 2-310) corresponds to the existing `StandardGetFile` function; the `CustomGetFilePreview` function (described on page 2-312) corresponds to the existing `CustomGetFile` function. Both of these new functions take the same parameters as their existing counterparts. See *Inside Macintosh: Files* for information about `StandardGetFile` and `CustomGetFile`.

The `StandardGetFilePreview` function uses the dialog box shown in Figure 2-36. The `CustomGetFilePreview` function can also use this dialog box, if you do not supply your own.

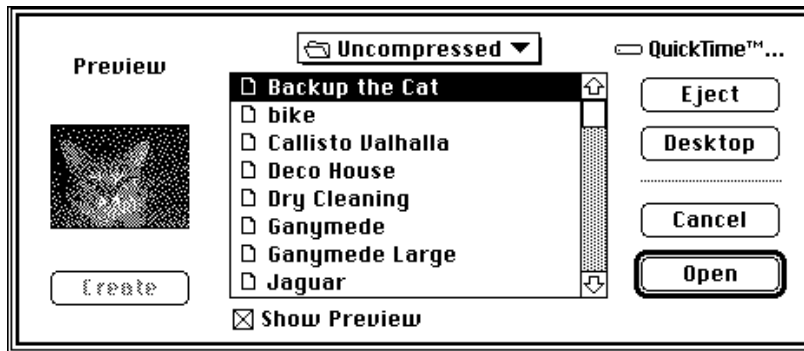
**Figure 2-36** `StandardGetFilePreview` or `CustomGetFilePreview` dialog box without preview



You use these new functions in place of the existing standard file functions whenever you want to allow the user to display previews during the Open dialog box. The user causes a file's preview to be displayed by selecting a file in the dialog box and clicking Show Preview. When the user does so, the functions display the preview for the file, as shown in Figure 2-37.



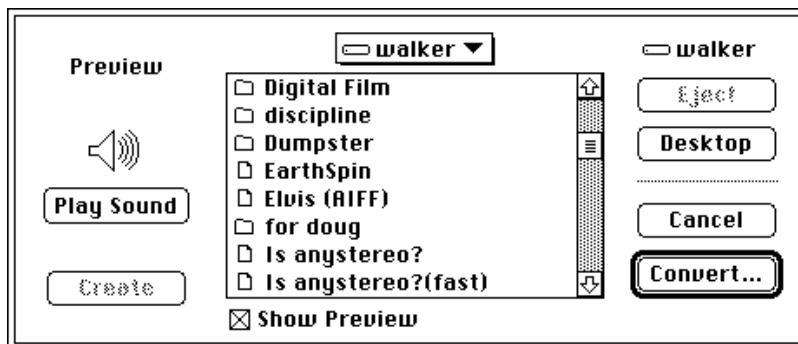
**Figure 2-37** StandardGetFilePreview or CustomGetFilePreview dialog box with preview



The preview portion of the dialog box is displayed only when the dialog box is showing a file's preview.

The SFGGetFilePreview, SFPGetFilePreview, StandardGetFilePreview, and CustomGetFilePreview functions allow the user to automatically convert files to movies if your application requests movies. If there is a file that can be converted into a movie file using a movie import component, then the file is shown in the Standard File dialog box in addition to any movies. When the user selects the file, the Open button changes to a Convert button. Figure 2-38 provides an example of this dialog box.

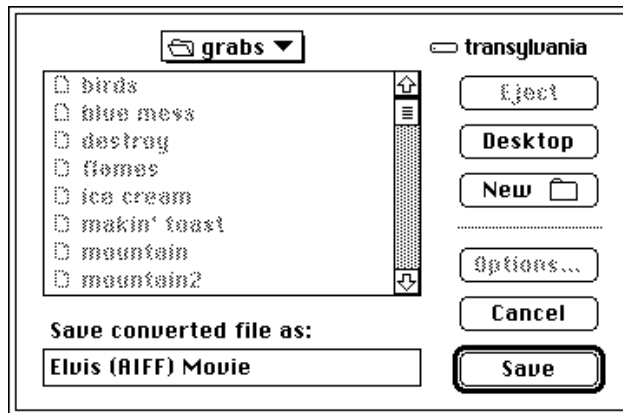
**Figure 2-38** Dialog box showing automatic file-to-movie conversion option



## Movie Toolbox

Choosing Convert displays a dialog box that allows the user to choose where the converted file should be saved. Figure 2-39 shows this dialog box.

**Figure 2-39** Dialog box for saving a movie converted from a file



When conversion is complete, the converted file is returned to the calling application as the movie that the user chose. If you want to disable automatic file conversion in your application, you must write a file filter function and pass it to the file preview display function you are using. Your file filter function must call the File Manager's `FSpGetFileInfo` function on each file that is passed to it to determine its actual file type. If the File System parameter block pointer passed to your file filter function indicates that the file type is `'Moov'`, and the actual type returned by `FSpGetFileInfo` is not `'Moov'`, then the file filter function will convert this file. If you do not wish a file to be displayed as a candidate for conversion, your file filter function should return a value of `true` when it is called for that file.

See “File Filter Functions” beginning on page 2-360 for comprehensive details on the interaction of application-defined file filter functions with the file preview display functions. For information on `FSpGetFileInfo`, see *Inside Macintosh: Files*.

## Customizing Your Interface in System 7

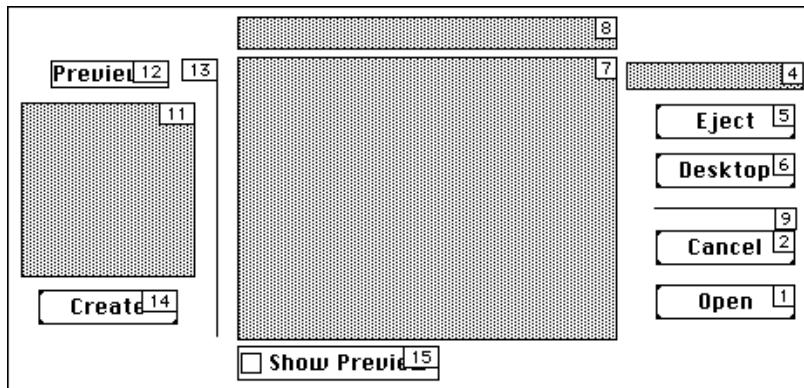
If your application requires it, you can customize the user interface for identifying files. The `CustomGetFilePreview` function allows you to specify a custom dialog box of any resource type with the `dlgID` parameter.

Figure 2-40 shows the standard dialog box used by `CustomGetFilePreview`. Your dialog box and dialog filter function must support at least these dialog items.

**Note**

Alter the dialog boxes only if necessary. Apple does not guarantee future compatibility if you use a customized dialog box. ♦

**Figure 2-40** Standard preview dialog box for CustomGetFilePreview



Items to the left of item 13 are visible only when previewing. If you want to define items that are visible only during a file preview, place them to the left of item 13 in your custom dialog box.

If your application defines a custom dialog box, be sure to include the following items in your dialog box definition:

```
enum
{
    /* dialog items to include in dialog box definition */
    sfpItemPreviewAreaUser      = 11, /* user preview area */
    sfpItemPreviewStaticText    = 12, /* static text preview */
    sfpItemPreviewDividerUser   = 13, /* user divider preview */
    sfpItemCreatePreviewButton   = 14, /* create preview button */
    sfpItemShowPreviewButton     = 15 /* show preview button */
};
```

## Using Application-Defined Functions

The Movie Toolbox allows your application to define functions that are invoked during specific operations. You can create a **progress function** that monitors the Movie Toolbox's progress on long operations, and you can create a **cover function** that allows your application to perform custom display processing.

See "Application-Defined Functions," which begins on page 2-354, for comprehensive details on these two types of functions.

## Movie Toolbox

Listing 2-14 shows two sample cover functions. Whenever a movie covers a portion of a window, the `MyCoverProc` function removes the covered region from the window's clipping region. When a movie uncovers a screen region, the `MyUncoverProc` function invalidates the region and adds it to the window's clipping region. By invalidating the region, this function causes the application to receive an update event, informing the application to redraw its window. The `InitCoverProcs` function initializes the window's clipping region and installs these cover functions.

---

**Listing 2-14** Two sample movie cover functions

```
pascal OSErr MyCoverProc (Movie aMovie,  RgnHandle changedRgn,
                          long refcon)
{
    CGrafPtr    mPort;
    GDHandle    mGD;

    GetMovieGWorld (aMovie, &mPort, &mGD);
    DiffRgn (mPort->clipRgn, changedRgn, mPort->clipRgn);
    return noErr;
}

pascal OSErr MyUnCoverProc (Movie aMovie,  RgnHandle changedRgn,
                            long refcon)
{
    CGrafPtr    mPort, curPort;
    GDHandle    mGD, curGD;

    GetMovieGWorld (aMovie, &mPort, &mGD);
    GetGWorld (&curPort, &curGD);
    SetGWorld (mPort, mGD);

    InvalRgn (changedRgn);
    UnionRgn (mPort->clipRgn, changedRgn, mPort->clipRgn);

    SetGWorld (curPort, curGD);
    return noErr;
}

void InitCoverProcs (WindowPtr aWindow,  Movie aMovie)
{
```

## Movie Toolbox

```

RgnHandle    displayBounds;
GrafPtr      curPort;

displayBounds = GetMovieDisplayBoundsRgn (aMovie);
if (displayBounds == nil) return;

GetPort (&curPort);
SetPort (aWindow);
ClipRect (&aWindow->portRect);
DiffRgn (aWindow->clipRgn, displayBounds, aWindow->clipRgn);
DisposeRgn( displayBounds );
SetPort (curPort);

SetMovieCoverProcs (aMovie, &MyUnCoverProc, &MyCoverProc, 0);
}

```

## Working With Movie Spatial Characteristics

The following section provides an example of how to create a track matte. Listing 2-15 provides an example of how to create a track matte. The `CreateTrackMatte` function adds an uninitialized, 8-bit-deep, grayscale matte to a track. The `UpdateTrackMatte` function draws a gray ramp rectangle around the edge of the matte and fills the center of the matte with black. (A ramp rectangle shades gradually from light to dark in smooth increments.)

---

### Listing 2-15 Creating a track matte

```

void CreateTrackMatte (Track theTrack)
{
    QDErr err;
    GWorldPtr aGW;
    Rect trackBox;
    Fixed trackHeight;
    Fixed trackWidth;
    CTabHandle grayCTab;

    GetTrackDimensions (theTrack, &trackWidth, &trackHeight);
    SetRect (&trackBox, 0, 0, FixRound (trackWidth),
            FixRound (trackHeight));
}

```

## Movie Toolbox

```

    grayCTab = GetCTable(40); /* 8 bit + 32 = 8 bit gray */
    err = NewGWorld (&aGW, 8, &trackBox, grayCTab,
                    (GDHandle) nil, 0);
    DisposeCTable (grayCTab);
    if (!err && (aGW != nil))
    {
        SetTrackMatte (theTrack, aGW->portPixMap);
        DisposeGWorld (aGW);
    }
}

void UpdateTrackMatte (Track theTrack)
{
    OSErr err;
    PixMapHandle trackMatte;
    PixMapHandle savePortPix;
    Movie      theMovie;
    GWorldPtr tempGW;
    CGrafPtr savePort;
    GDHandle saveGDevice;
    Rect      matteBox;
    short     i;

    theMovie = GetTrackMovie (theTrack);
    trackMatte = GetTrackMatte (theTrack);
    if (trackMatte == nil)
    {
        /* track doesn't have a matte, so give it one */
        CreateTrackMatte (theTrack);
        trackMatte = GetTrackMatte (theTrack);
        if (trackMatte == nil)
            return;
    }
}

```

## Movie Toolbox

```

GetGWorld (&savePort, &saveGDevice);
matteBox = (**trackMatte).bounds;
err = NewGWorld(&tempGW,
                (**trackMatte).pixelSize, &matteBox,
                (**trackMatte).pmTable, (GDHandle) nil, 0);
if (err || (tempGW == nil)) return;

SetGWorld (tempGW, nil);
savePortPix = tempGW->portPixMap;
LockPixels (trackMatte);
SetPortPix (trackMatte);

/* draw a gray ramp rectangle around the edge of the matte */
for (i = 0; i < 35; i++)
{
    RGBColor aColor;
    long      tempLong;

    tempLong = 65536 - ((65536 / 35) * (long)i);
    aColor.red = aColor.green = aColor.blue = tempLong;
    RGBForeColor(&aColor);
    FrameRect (&matteBox);
    InsetRect (&matteBox, 1, 1);
}

/* fill the center of the matte with black */
ForeColor (blackColor);
PaintRect (&matteBox);

SetPortPix (savePortPix);
SetGWorld (savePort, saveGDevice);
DisposeGWorld (tempGW);

UnlockPixels (trackMatte);
SetTrackMatte (theTrack, trackMatte);

DisposeMatte (trackMatte);
}

```