QuickTime Movie Basics

2006-01-10

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

Simultaneously published in the United States and Canada.

Contents

Figures, Tables, and Listings

Introduction to QuickTime Movie Basics

This book introduces you to some of the basic concepts you need to understand when working with QuickTime movies.

Note: This book replaces four previously separate Apple documents: "Movie Toolbox: Editing," "Movie Toolbox: Application-Defined Functions," "Movie Toolbox: Data Types," and "Movie Toolbox: Saving Movies."

You need to read this book if you are going to work with QuickTime movies.

Organization of This Document

This book consists of the following chapters:

- Editing Movies (page 11) describes the functions of the movie toolbox that your application will use to edit movies and tracks.
- Saving Movies (page 19) tells you how to save movies into movie files.
- Overview of Application-Defined Functions (page 23) describes the application-defined functions used with the QuickTime Movie Toolbox.
- Working With Application-Defined Functions (page 27) shows how to implement functions that are invoked during specific operations. A sample cover function is provided.
- Creating Tracks and Media Structures (page 29) describes the functions your application can use to create and dispose of tracks and media structures.
- Working With Media Samples (page 35) describes the functions your application can use to get information about a movie's sample data.
- Understanding QuickTime Atoms (page 39) discusses QuickTime atoms, a basic structure for storing information in QuickTime, and also describes the functions used to create, dispose of, read from, and store to QuickTime Atom Containers.
- Timecode Media Handler Functions (page 51) discusses the functions and structures that allow you to use the timecode media handler, and tells you how to create a timecode track.
- Locating Tracks, Saving Movies, and Modifying Movie Properties (page 57) discusses functions you can use to locate a movie's tracks and media structures, save movies, capture and restore the edit state of a movie, and modify movie properties.
- Matrix Functions (page 65) describes the functions that allow you to work with transformation matrices.
- Movie Toolbox Constants, Data Types, and Functions (page 69) describes some of the constants, data types, and functions in the Movie Toolbox that you can use in your application development.

See Also

The following Apple books cover related aspects of QuickTime programming:

- *QuickTime Overview* gives you the starting information you need to do QuickTime programming.
- *QuickTime Movie Creation Guide* describes some of the different ways your application can create a new QuickTime movie.
- *QuickTime Movie Internals Guide* covers some of the technology present inside QuickTime movies, including time management, modifier tracks, access keys, posters, and movie and file previews.
- *QuickTime Guide for Windows* provides information specific to programming for QuickTime on the Windows platform.
- *QuickTime API Reference* provides encyclopedic details of all the functions, callbacks, data types and structures, atom types, and constants in the QuickTime API.

Editing Movies

This chapter describes the functions of the movie toolbox that your application will use to edit movies and tracks. You will need to read this chapter if your application provides editing functions beyond those that are built into the movie controller.

The Movie Toolbox provides a set of high-level functions that allow you to edit movies. This chapter describes these high-level editing functions. These functions work with a movie's current selection. The current selection is defined by a starting time and a duration.

The Movie Toolbox also provides functions that allow you to edit movie segments. Those functions are described in Low-Level Movie-Editing Functions (page 16).

The movies created by these functions contain references to the data in the source movie. Because the new movies contain references and not data, they are small and easily moved to and from the scrap. If you delete the movie that contains the data, the data references in the new movies are no longer valid and the new movies cannot be played. Therefore, before you delete the original movie, you should call the `FlattenMovie` function for each of the new movies. This function copies the data into each of the new movies, eliminating the data references.

Note that the Movie Toolbox does not always copy empty tracks from the source movie to the movies that are created by these functions. Specifically, the Movie Toolbox preserves the empty tracks until you paste or add the selection into the destination movie. At that time, the Movie Toolbox removes the empty tracks from the selection. In addition, if a track in the source movie has trailing empty space, the Movie Toolbox removes that empty space from the track when it is copied into the new movie. Therefore, if you want to add a segment beyond the end of a movie, you insert the space when you insert the new segment using the `InsertMovieSegment` function.

The Movie Toolbox allows you to paste different data types into a movie. For example, QuickDraw pictures and standard sound data can be pasted directly into a movie. If you are using the movie controller component, you do not need to use these functions to paste different data types into a movie. If you are calling the Movie Toolbox directly to do editing, you should use the functions described in this section.

Handling Media Sample References

You could always use `GetMediaSampleReference` to access samples in a movie one at a time. But QuickTime also has `GetMediaSampleReferences` (note that this is the plural form of the `GetMediaSampleReference` function), which you can use to obtain information about groups of samples. In addition you can use `AddMediaSampleReferences` to work with groups of samples that have already been added to a movie.

Manipulating Media Input Maps

The Movie Toolbox contains two functions for maintaining media input maps: `GetMediaInputMap` and `SetMediaInputMap`.

Each track has particular attributes such as size, position, and volume associated with it. The media input map of that track describes where the variable parameters are stored so that modifier tracks know where to send their data. When a track is copied, its input map is also copied. `CopyTrackSettings` also transfers the media input map.

Undo for Movies

The Movie Toolbox provides functions that allow you to capture and restore the edit state of a movie. An **edit state** contains information that completely defines a movie's content at the time you create the edit state. It is, in essence, a checkpoint in the edit session. You can manage a movie's edit states in order to implement an undo capability for editing movies. For example, you can capture a movie's edit state before performing an editing operation, such as a cut, and later restore the old state. You can have several movie edit states obtained at different times during an editing session, and restore to any one of them at any time. In this manner, you can provide a multilevel undo capability. This section describes the Movie Toolbox functions that work with edit states.

Note that a movie's edit state does not save everything about a movie. Most important, the edit state does not contain information about the movie's spatial characteristics. For example, the edit state does not store the current boundary rectangle or clipping region. Consequently, edit states are best suited to supporting undo operations involving movie content, including track creation and removal. You can use other Movie Toolbox functions to support undo operations for movie characteristics. See Functions That Modify Movie Properties (page 12) to learn more about these functions.

You can use the `NewMovieEditState` function to capture a movie's edit state. Use the `UseMovieEditState` to restore the movie to its condition according to a previous edit state. Your application must dispose of an edit state by calling the `DisposeMovieEditState` function. You must dispose of a movie's edit states before you dispose of the movie.

Functions That Modify Movie Properties

The Movie Toolbox provides a number of functions that allow applications to edit existing movies or to create the contents of new movies. This section describes those functions. It has been divided into the following topics:

- Working With Display Characteristics (page 13) describes a number of functions that allow you to work with the display characteristics of movies
- Working With Sound Volume (page 14) discusses the functions that your application can use to work with the sound volume of a movie or a track
- Locating a Movie's Tracks and Media Structures (page 14) describes the functions that allow your application to find tracks that are associated with a movie

- Determining Movie Creation and Modification Time (page 15) discusses the functions that you can use to determine when a movie was created or last changed

Working With Display Characteristics

The Movie Toolbox provides a number of functions that allow your application to determine and change the display characteristics of movies and tracks. These functions are discussed in the following sections. Before using any of these functions, you should be familiar with the way in which the Movie Toolbox displays movies.

You can use the `SetMovieGWorld` and `GetMovieGWorld` functions to work with a movie's graphics world.

Your application can work with a movie's matrix by calling the `GetMovieMatrix` and `SetMovieMatrix` functions, and it can work with a track's matrix with the `GetTrackMatrix` and `SetTrackMatrix` functions. Then you can perform operations on matrices with the Movie Toolbox's matrix functions.

The following functions affect the displayed movie and its tracks in the final display coordinate system. The `SetMovieGWorld` and `GetMovieGWorld` functions let you work with a movie's display destination. The `GetMovieBox` and `SetMovieBox` functions allow you to work with a movie's boundary rectangle and its associated transformations. Alternatively, you can use the `GetMovieMatrix` and `SetMovieMatrix` functions to work directly with a movie's transformation matrix. The `GetMovieDisplayBoundsRgn` function determines a movie's boundary region at the current movie time. On the other hand, the `GetMovieSegmentDisplayBoundsRgn` function determines a movie's boundary region over a specified time segment. You can use the `GetMovieDisplayClipRgn` and `SetMovieDisplayClipRgn` functions to work with a movie's display clipping region.

The `GetTrackDisplayBoundsRgn` and `GetTrackSegmentDisplayBoundsRgn` functions determine a track's final boundary region. You can use the `GetTrackLayer` and `SetTrackLayer` functions to control the drawing order of tracks within a movie.

A number of functions affect a movie's display boundaries before any display transformations. These functions operate in the movie's display coordinate system. You can use the `GetMovieClipRgn` and `SetMovieClipRgn` functions to work with a movie's clipping region; that is, the clipping region that is applied before the movie display transformation. Use the `GetMovieBoundsRgn` function to determine a movie's boundary region at the current movie time.

Use the `GetTrackMovieBoundsRgn` function to work with a track's boundary region after matrix transformations have placed the track into the movie's display system. The `SetTrackMatrix` and `GetTrackMatrix` functions let you define a track's matrix transformations.

The Movie Toolbox provides several functions that affect a track's display boundaries. These functions operate in the track's display coordinate system before any other display transformations are applied. The `GetTrackDimensions` and `SetTrackDimensions` functions allow you to establish a track's coordinate system and to establish a track's source rectangle.

Note: A track's source rectangle defines the coordinate system of the track. You specify the dimensions of the rectangle by providing the coordinates of the lower-right corner of the rectangle. The Movie Toolbox sets the upper-left corner to (0,0) in the track's coordinate system.

You can use the `GetTrackBoundsRgn` function to determine a track's boundary region. The `GetTrackClipRgn` and `SetTrackClipRgn` functions let you work with a track's clipping region. You can use the `GetTrackMatte` and `SetTrackMatte` functions to establish a track's matte. The `DisposeMatte` function allows you to dispose of a matte once you are finished with it.

Working With Sound Volume

The Movie Toolbox allows you to set the sound volume of movies and tracks. Track volumes allow tracks within a movie to have different volumes. A track's volume is scaled by the movie's volume to produce the track's final volume. Furthermore, the movie's volume is scaled by the sound volume that is returned by the Sound Manager's `GetSoundVol` routine. Thus, the user can control the overall volume from the Sound control panel.

Volume values range from -1.0 to 1.0. Higher values translate to louder volume. Negative values indicate muted volume. That is, the Movie Toolbox does not play any sound for movies or tracks with negative volume settings, but the original volume level is retained as the absolute value of the volume setting. Therefore, if you want to toggle the current state of the volume, you can invert the sign of the current volume setting, as shown here:

```
SetMovieVolume ( theMovie, GetMovieVolume(theMovie) );
```

You can use the `GetMovieVolume` and `SetMovieVolume` functions to work with a movie's volume.

The `GetTrackVolume` and `SetTrackVolume` functions allow you to work with a track's volume.

Locating a Movie's Tracks and Media Structures

The Movie Toolbox provides a set of functions that help your application locate a movie's tracks and media structures. This section describes these functions.

The Movie Toolbox identifies a movie's tracks in two ways. First, every track in a movie has a unique ID value. This ID value is unique throughout the life of a movie, even after it has been saved. That is, no two tracks of a movie ever have the same ID, and no ID value is ever reused. Second, a movie's current tracks may be identified by their index value. Index values always range from 1 to the number of tracks in the movie. Track indexes provide a convenient way to access each track of a movie.

There are several functions that allow you to find a movie's tracks. You can use the `GetMovieTrackCount` function to determine the number of tracks in a movie. Use the `GetMovieTrack` function to obtain the track identifier for a specific track, given its ID. The `GetMovieIndTrack` function lets you obtain a track's identifier, given its track index.

You can obtain a track's ID value given its track identifier by calling the `GetTrackID` function.

You can determine the movie that contains a track by calling the `GetTrackMovie` function.

The `GetTrackMedia` function enables you to find a track's media. Conversely, you can find the track that uses a media by calling the `GetMediaTrack` function.

Determining Movie Creation and Modification Time

The Movie Toolbox maintains two timestamps in every movie, track, and media. One timestamp, the creation date, indicates the date and time when the item was created. The other, the modification date, contains the date and time when the item was last changed and saved. The timestamp value is in the same format as Macintosh file system creation and modification times; that is, the timestamp indicates the number of seconds since midnight, January 1, 1904.

The Movie Toolbox provides a number of functions that allow your application to retrieve the creation and modification date information from movies, tracks, and media structures.

You can use the `GetMovieCreationTime` and `GetMovieModificationTime` functions to work with movie creation and modification dates.

You can use the `GetTrackCreationTime` and `GetTrackModificationTime` functions to retrieve a track's creation and modification dates.

Your application can call the `GetMediaCreationTime` and `GetMediaModificationTime` functions to get a media's creation and modification dates.

Editing Tracks

The Movie Toolbox provides a number of functions that allow your application to perform editing operations on tracks. These functions work with track segments (pieces of a track that are defined by a starting time and duration) and therefore give you a great deal of control over the editing process. These functions are similar to the low-level editing functions for movies that were described earlier in this chapter. However, these functions may copy movie data, if required by the operation.

When you edit a track you may change the duration of the movie that contains that track.

The `CopyTrackSettings` function lets you copy certain important settings from one track to another.

You can use the `InsertTrackSegment` function to copy a segment from one track to another, by reference or by moving data, or to copy a segment within a track. The `InsertTrackEmptySegment` function allows you to insert an empty segment into a track.

You can use the `InsertMediaIntoTrack` function to insert a media into a track.

Your application can delete a segment from a track by calling the `DeleteTrackSegment` function.

You can change a segment's duration by calling the `ScaleTrackSegment` function. This function stretches or shrinks the segment to accommodate a specified duration.

You can use the `GetTrackEditRate` function to determine the rate of the track edit of a specified track at an indicated time.

Undo for Tracks

The Movie Toolbox provides functions that allow you to capture and restore the edit state of a track. As with the functions that manipulate a movie's edit state, you can manage a track's edit states in order to implement an undo capability for track editing. For example, you can capture a track's edit state before performing an editing operation, such as a cut, and later restore the old state. You can have several track edit states obtained at different times during an editing session, and you can restore to any one of them at any time. In this manner, you can provide a multilevel undo capability. This section describes the Movie Toolbox functions that work with track edit states.

Note that a track's edit state does not save everything about the track. Most important, the edit state does not contain information about track spatial characteristics. For example, the edit state does not store the current clipping region. Consequently, edit states are best suited to supporting undo operations involving track content. You can use other Movie Toolbox functions to support undo operations for track characteristics. See Functions That Modify Movie Properties (page 12) to learn more about these functions.

You can use the `NewTrackEditState` function to capture a track's edit state. Use the `UseTrackEditState` function to restore the track to its condition according to a previous edit state. Your application can dispose of an edit state by calling the `DisposeTrackEditState` function.

Selection and Scrap Functions

To get and change a movie's current selection, your application can call the `GetMovieSelection` and `SetMovieSelection` functions.

Your application can work with a movie's current selection by calling the `CutMovieSelection`, `CopyMovieSelection`, `PasteMovieSelection`, `ClearMovieSelection`, and `AddMovieSelection` functions.

The `PutMovieOnScrap` and `NewMovieFromScrap` functions enable your application to work with movies that are on the scrap.

The `IsScrapMovie` function examines the system scrap to determine whether it can translate any of the data into a movie. The `PasteHandleIntoMovie` takes the contents of a specified handle, together with its type, and pastes it into a movie. `PutMovieIntoTypedHandle` takes a movie (or a single track from within a movie) and converts it into a handle.

Low-Level Movie-Editing Functions

The Movie Toolbox provides a number of functions that allow your application to perform low-level editing operations on movies. These functions work with movie segments (pieces of a movie that are defined by a starting time and duration) and therefore give you a great deal of control over the editing process. These functions never copy the movie data; rather, they work with references to the movie's data. Editing Movies (page 11) discusses the Movie Toolbox functions that allow you to edit movies by working with the current selection.

You can use the `CopyMovieSettings` function to copy certain important settings from one movie to another.

You can use the `InsertMovieSegment` function to copy a segment from one movie to another. Use the `InsertMovieEmptySegment` function to insert an empty segment into a movie.

Your application can delete a segment from a movie by calling the `DeleteMovieSegment` function.

You can change a segment's duration by calling the `ScaleMovieSegment` function. This function stretches or shrinks the segment to accommodate a specified duration.

Summary of Editing Functions

This section lists the functions described earlier in this chapter.

- Editing Movies
  - `PutMovieOnScrap`
  - `NewMovieFromScrap`
  - `IsScrapMovie`
  - `SetMovieSelection`
  - `GetMovieSelection`
  - `CutMovieSelection`
  - `CopyMovieSelection`
  - `PasteMovieSelection`
  - `AddMovieSelection`
  - `ClearMovieSelection`
  - `PasteHandleIntoMovie`
  - `PutMovieIntoTypedHandle`
  - `PasteHandleIntoMovie`
- Undo for Movies
  - `NewMovieEditState`
  - `UseMovieEditState`
  - `DisposeMovieEditState`
- Low Level Movie Editing Functions
  - `InsertMovieSegment`
  - `InsertEmptyMovieSegment`
  - `DeleteMovieSegment`
  - `ScaleMovieSegment`
  - `CopyMovieSettings`

- Editing Tracks
  - `InsertTrackSegment`
  - `InsertEmptyTrackSegment`
  - `InsertMediaIntoTrack`
  - `DeleteTrackSegment`
  - `ScaleTrackSegment`
  - `CopyTrackSettings`
  - `GetTrackEditRate`
  - `AddEmptyTrackToMovie`
- Undo for Tracks
  - `NewTrackEditState`
  - `UseTrackEditState`
  - `DisposeTrackEditState`
- Locating a Movie's Tracks and Media Structures
  - `GetMovieTrackCount`
  - `GetMovieIndTrack`
  - `GetMovieTrack`
  - `GetTrackID`
  - `GetTrackMovie`
  - `GetTrackMedia`
  - `GetMediaTrack`
  - `GetMovieIndTrackType`

You can find details of these functions in the *QuickTime API Reference*.

Saving Movies

This chapter briefly describes saving movies into movie files. Saving a movie into a new movie file and saving movie contents to an existing movie file are both discussed.

Your application can gain access to existing movies with either the `NewMovieFromFile` function or the `NewMovieFromDataFork` function. Once you have loaded the movie, your application uses the functions that are described in this section to save any changes you have made to the movie.

Movie File Functions

The Movie Toolbox provides a set of functions that allow your application to create, access, and convert movie files, which files contain data for QuickTime movies. You can also use the Movie Toolbox to load movies into memory, in preparation for working with the movie. These functions differ based on where the movie is stored.

The following functions are used when saving a movie to a file.

- `HasMovieChanged`
- `ClearMovieChanged`
- `AddMovieResource`
- `UpdateMovieResource`
- `RemoveMovieResource`
- `PutMovieIntoHandle`
- `FlattenMovie`
- `FlattenMovieData`
- `NewMovieFromDataFork`
- `PutMovieIntoDataFork`

Before your application can play a movie, you must first open the file that contains the movie. Your application can use the `OpenMovieFile` function to open a movie file. Once you are done with the file, your application releases the file by calling the `CloseMovieFile` function. Your application can create and open a new movie file by calling the `CreateMovieFile` function. Your application can delete a movie file by calling the `DeleteMovieFile` function.

You can use the `NewMovie` function to create a new empty movie. If your application is loading a movie from an existing file, use either the `NewMovieFromFile` function or the `NewMovieFromDataFork` function. The `NewMovieFromFile` function works with the file reference number you obtain from the `OpenMovieFile` function. The `NewMovieFromDataFork` function works with movies stored in your document file's data fork. Your application can then use the functions described in Saving Movies (page 19) to load and store movies.

You can use the `ConvertFileToMovieFile` function to specify an input file and convert it to a movie file. The `ConvertMovieToFile` takes a specified movie (or a single track within that movie) and converts it into an output file.

You can use the `AddMovieResource` function to add a new movie resource to a movie file. Your application can also use this function to save a movie that it created. You can use the `UpdateMovieResource` function to replace an existing movie resource in a movie file. You can remove a movie resource by calling the `RemoveMovieResource` function.

The movie resources that your application creates with the `AddMovieResource` and `UpdateMovieResource` functions may contain references to movie data. These references identify the data that constitute the movie. However, the movie data can be stored outside of the movie file. If you want to create a movie file that contains all of its movie data, use the `FlattenMovie` or `FlattenMovieData` function. These functions can also be used to store the movie data in the movie file's data fork, or to interleave the media data to optimize performance.

The `PutMovieIntoHandle` function places a QuickTime movie into a handle. You can then convert the movie into specialized data formats.

The `HasMovieChanged` and `ClearMovieChanged` functions allow your application to work with the movie changed flag that is maintained by the Movie Toolbox. You can use this flag to determine whether a movie has been changed.

The movie changed flag indicates whether you have changed the movie. Such actions as editing the movie, adding samples to a media, or changing a data reference cause the flag to indicate that the movie has changed. There are several operations that the movie changed flag does not reflect, including changing the volume, rate, or time settings for the movie. These settings change frequently when a movie is played. Your application must monitor these settings itself.

The Movie Toolbox also supplies functions for storing and retrieving movies that are stored in the data fork of a file. These functions provide robust data reference resolution and improve low memory performance. The `NewMovieFromDataFork` function enables you to retrieve a movie that is stored anywhere in the data fork of a file. You can use the `PutMovieIntoDataFork` function to store an atom version of a specified movie in the data fork of a file.

Once you are finished working with a movie, you should release the resources used by the movie by calling the `DisposeMovie` function.

Movie Files

The Movie Toolbox allows you to save movies in movie files. Movie files have a file type of `'MooV'Movie` Toolbox. Typically, the movie itself is stored in the resource fork of the movie file. The movie can also be stored in data fork, for use on systems that do not support resource forks. 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 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. When you are done with the file, you close it by calling the `CloseMovieFile` function. 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. You then load the movie from that file by calling the `NewMovieFromFile` function. Alternatively, you can use the `NewMovieFromHandle` function. 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. If you want to keep the old movie, create a new movie by calling the `AddMovieResource` function. A movie file may contain more than one movie resource. You should then close the movie file by calling the `CloseMovieFile` function.

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. You can set the flag to `false` by calling the `ClearMovieChanged` function.

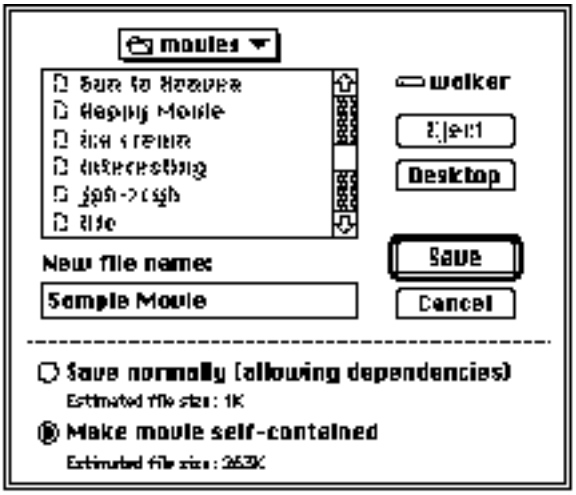
The Movie Toolbox provides two functions for deleting movies: `DeleteMovieFile` and `RemoveMovieResource`. Use `DeleteMovieFile` to delete a movie file. Use `RemoveMovieResource` to delete a movie from a movie file. Don't use the corresponding standard Macintosh Toolbox routines (`FSpDelete` and `RmveResourceMovie`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

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 is often desirable when creating a version of a movie that will be moved to another computer. You can use the `FlattenMovie` or `FlattenMovieData` functions to resolve all of the data references and create a self-contained movie.

The Movie Toolbox also accommodates operating systems that do not recognize files that contain more than one fork. You can create a movie file that contains the movie and all of its data in the data fork of the movie file. You can then use that file on operating systems that do not recognize resource forks. You can use the `FlattenMovie` or `FlattenMovieData` functions to put the movie in the data fork at the same time it creates a self-contained movie file. This would be the usual approach when creating a movie on a Macintosh computer that you want to store on a Unix web server. You can also create a single-fork movie file by calling the `CreateMovieFile` function with the flag `createMovieFileDontCreateResFileMovie` Toolbox. This would be the usual approach when creating movies using QuickTime for Windows.

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 2-1. 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 2-1 A sample movie Save As dialog box



Overview of Application-Defined Functions

This section describes the functions your application can provide when interacting with the Movie Toolbox, including some functions that your application must provide to make use of certain services. These functions allow you to monitor, customize, and extend the behavior of the Movie Toolbox by executing functions you create at various points in the operation of the Movie Toolbox.

Progress Functions

Some Movie Toolbox functions can take a long time to execute. For example, creating a movie file that contains all of its data may be quite an involved process for a movie that has many large media structures. During these operations, your application should give the user some indication of the progress of the task. The Movie Toolbox allows you to monitor its progress on long operations with a progress function.

The Movie Toolbox calls your progress function at regular intervals during long operations. The Movie Toolbox determines whether to call your function based on the duration of the operation; your function will not be called unnecessarily. When it calls your function, the Movie Toolbox provides information about the operation that is underway and its relative completion. You can use this information to display an informational dialog box to the user.

You assign a progress function to a movie by calling the `SetMovieProgressProc` function. You should assign your progress function when you open the movie; the Movie Toolbox will call your function when it is appropriate to do so. One progress function may support more than one movie. When the Movie Toolbox calls your function, it provides you with the movie identifier so that you can discriminate between various movies.

Cover Functions

The Movie Toolbox allows your application to perform custom processing whenever one of your movies covers a screen region or reveals a region that was previously covered. You perform this processing using cover functions.

There are two types of cover functions: those that are called when your movie covers a screen region, and those that are called when your movie uncovers a screen region, revealing a region that was previously covered. You can use a cover function to detect when a movie changes size.

Cover functions that are called when your movie covers a screen region are responsible for erasing the region; you may choose to save the hidden region in an offscreen buffer. Cover functions that are called when your movie reveals a hidden screen region must redisplay the hidden region.

The Movie Toolbox sets the graphics world before it calls your cover function. Your function must not change the graphics world.

The Movie Toolbox provides default cover functions. When your movie uncovers a region, the default function that is called erases the movie's image by displaying the graphics port's background color and pattern. You can set the port's characteristics by calling the `SetMovieGWorld` function. When your movie covers a region, the default function that is called does nothing.

Use the `SetMovieCoverProcs` function to set both types of cover functions.

Error-Notification Functions

The Movie Toolbox lets your application perform custom error notification. Your application must identify its custom error-notification function to the Movie Toolbox. Error-notification functions are particularly helpful when you are debugging your program.

The `SetMoviesErrorProc` function allows you to identify your application's error-notification function in the `errProc` parameter.

Movie Callout Functions

The `PlayMoviePreview` function plays a movie's preview. You provide a pointer to a movie callout function in the callOutProc parameter.

The Movie Toolbox calls your movie callout function repeatedly while the movie preview is playing. You can use this function to stop the preview. If you do not want to assign a function, set the callOutProc parameter to `nil`.

File Filter Functions

A file filter function filters the files that are displayed to the user in a dialog box. You specify this function in the `fileFilter` parameter of the `SFGetFilePreview`, `StandardGetFilePreview`, and `CustomGetFilePreview` routines. If this parameter is not `nil`, `SFGetFilePreview` calls the function for each file to determine whether to display the file to the user. The `SFGetFilePreview` function supplies you with the information it receives from the File Manager's GetFileInfo routine.

Custom Dialog Functions

A dialog hook function handles user selections in a dialog box. A custom dialog function lets you support the template in the custom dialog template that you specified with the CustomGetFilePreview routine. This function corresponds to the File Manager's `CustomGetFile` routine.

You specify your dialog function in the dlgHook parameter of `CustomGetFilePreview`. You can use this parameter to support a custom dialog box function you have supplied by specifying a dialog template resource in your resource file. You specify the dialog template's resource ID with the `dlgID` parameter. If you are not supplying a custom dialog function, set this parameter to `nil`.

Modal-Dialog Filter Functions

The `CustomGetFilePreview` routine presents an Open dialog box to the user and allows the user to view file previews. This function differs from `StandardGetFilePreview` in that you can provide a custom dialog template and functions to support your template. This function corresponds to the existing `CustomGetFile` routine.

You specify your modal-dialog filter function in the `filterProc` parameter. Your modal-dialog filter function gives you greater control over the interface presented to the user.

Note: A modal-dialog filter function controls events closer to their source by filtering the events received from the Event Manager. The Standard File Package itself contains an internal modal-dialog filter function that maps keypresses and other user input onto the equivalent dialog box items. If you also want to process events at this level, you can supply your own filter function.

Standard File Activation Functions

The `CustomGetFilePreview` function presents an Open dialog box to the user and allows the user to view file previews. This function differs from the `StandardGetFilePreview` function in that you can provide a custom dialog template and functions to support your template. The `CustomGetFilePreview` function corresponds to the File Manager's `CustomGetFile` routine.

You specify your activation function in the `activateProc` parameter. An activation function controls the highlighting of any items whose shape is known only by your application.

Callback Event Functions

The `CallMeWhen` function schedules a callback event. You specify the callback event in the `callBackProc` parameter.

Text Functions

You can use the MyTextProc function described in this section to pass a handle to a specified sample containing formatted text, along with the movie in which the text is being displayed, a pointer to a flag variable, and your reference constant. You specify the desired operations on the text and return an indication of whether you want to display the text in the `displayFlag` parameter.

Working With Application-Defined Functions

The Movie Toolbox allows your application to define functions that are invoked during specific operations. For example, 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.

Sample Cover Functions

Listing 4-1 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 4-1 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)
{
    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);
}
```

Creating Tracks and Media Structures

This book describes the functions your application can use to create and dispose of tracks and media structures. These functions are used when creating movies or when editing movies at the track level.

- `NewMovieTrack`
- `DisposeMovieTrack`
- `NewTrackMedia`
- `DisposeTrackMedia`

The Movie Toolbox provides several functions that allow your application to create new movie tracks and media structures and to dispose of existing tracks and media structures. You use these functions when you are creating a new movie or when you are editing an existing movie.

You can use the `NewMovieTrack` function to create a new track for a specified movie. Conversely, you can use the `DisposeMovieTrack` function to dispose of an existing track.

Your application can create a new media for a track by calling the `NewTrackMedia` function. You can use the `DisposeTrackMedia` function to dispose of an existing media.

Working With Track References

Track references allow you to relate tracks to one another. For example, this can help you identify the text track that contains the subtitles for a movie's audio track and relate that text track to a particular audio track.

About Track References

Although QuickTime has always allowed the creation of movies that contain more than one track, it has not been able to specify relationships between those tracks. **Track references** are a new feature of QuickTime that allow you to relate a movie's tracks to one another. The QuickTime track-reference mechanism supports many-to-many relationships. That is, any movie track may contain one or more track references, and any track may be related to one or more other tracks in the movie.

Track references can be useful in a variety of ways. For example, track references can be used to relate timecode tracks to other movie tracks. (See Timecode Media Handler (page 30) for more information about timecode tracks.) You might consider using track references to identify relationships between video and sound tracks: identifying the track that contains dialog and the track that contains background sounds, for example. Another use of track references is to associate one or more text tracks that contain subtitles with the appropriate audio track or tracks.

Track references are also used to create chapter lists, as described below.

Every movie track contains a list of its track references. Each track reference identifies another, related track. That related track is identified by its track identifier. The track reference itself contains information that allows you to classify the references by type. This type information is stored in an `OSType` data type. You are free to specify any type value you want. Note, however, that Apple has reserved all lowercase type values.

You may create as many track references as you want, and you may create more than one reference of a given type. Each track reference of a given type is assigned an index value. The index values start at 1 for each different reference type. The toolbox maintains these index values so that they always start at 1 and count by 1.

Track Reference Functions

This section describes the functions that manipulate track references. Track references define relations between tracks. For example, a timecode track may be related to several other tracks, or a text track may contain subtitles for a particular audio track.

- `AddTrackReference`
- `DeleteTrackReference`
- `SetTrackReference`
- `GetTrackReference`
- `GetNextTrackReferenceType`
- `GetTrackReferenceCount`

The `AddTrackReference` function allows you to relate one track to another. The `DeleteTrackReference` function removes that relationship. The `SetTrackReference` and `GetTrackReference` functions allow you to modify an existing track reference so that it identifies a different track. The `GetNextTrackReferenceType` and `GetTrackReferenceCount` functions allow you to scan all of a track's track references.

Timecode Media Handler

Timecode tracks allow you to store external timecode information, such as SMPTE timecodes, in your QuickTime movies. QuickTime now provides a new timecode media handler that interprets the data in these tracks.

Chapter Lists

A **chapter list** provides a set of named entry points into a movie, allowing the user to jump to a preselected point in the movie from a convenient pop-up list.

The movie controller automatically recognizes a chapter list and will create a pop-up list from it. When the user makes a selection from the pop-up, the controller will jump to the appropriate point in the movie.

Note: If the movie is sized so that the controller is too narrow to display the chapter names, the pop-up list will not appear.

To create a chapter list, you must create a text track with one sample for each chapter. The display time for each sample corresponds to the point in the movie that marks the beginning of that chapter. You must also create a track reference of type `'chap'` from an enabled track of the movie to the text track. It is the `'chap'` track reference that makes the text track into a chapter list. The track containing the reference can be of any type (audio, video, mpeg, etc.), but it must be enabled for the chapter list to be recognized.

Given an enabled track `myVideoTrack`, use `AddTrackReference` to create the chapter reference:

```
AddTrackReference( myVideoTrack, theTextTrack,
    kTrackReferenceChapterList, &addedIndex );
```

`kTrackReferenceChapterList` is defined in `Movies.h`. It has the value `'chap'`.

The text track which constitutes the chapter list does not need to be enabled, and normally is not. If it is enabled, the text track will be displayed as part of the movie, just like any other text track, in addition to functioning as a chapter list.

If more than one enabled track includes a `'chap'` track reference, QuickTime will use the first chapter list that it finds.

Working With Data References

This section describes the functions used to work with data references. Media structures point to their actual sample data using data references. For sound and video media, data references identify the files that contain the data. Media handlers also use data references in order to manipulate media data.

- `AddMediaDataRef`
- `SetMediaDataRef`
- `GetMediaDataRef`
- `GetMediaDataRefCount`

Media structures identify how and where to find their sample data by means of data references. For sound and video media, **data references** identify files that contain media data; the media data is stored in the data forks of these files. Media handlers use these data references in order to manipulate media data. A single media may contain one or more data references.

Each data reference contains type information that identifies how the reference is specified. Most QuickTime data references use alias information to locate the corresponding files. The type value for data references that use aliases is `'alis'`.

The Movie Toolbox identifies a media's data references with an index value. Index values always range from 1 to the number of references in the media. Data reference indexes provide a convenient way to access each reference in a media.

The Movie Toolbox provides a set of functions that allow you to work with data references. This section describes those functions.

You can use the `GetMediaDataRef` function to retrieve information about a media's data reference. You can add a data reference to a media by calling the `AddMediaDataRef` function. The `SetMediaRef` function lets you change which file a specified media associates with its data storage.

Your application can determine the number of data references in a media by calling the `GetMediaDataRefCount` function.

Alternate Tracks

QuickTime contains functions your application can use to work with alternate tracks. Alternate tracks are used to create a single movie that can, for example, play back with different language audio tracks in different countries, or with different quality audio or video on different model computers.

- `SetMovieLanguage`
- `SelectMovieAlternates`
- `SetAutoTrackAlternatesEnabled`
- `SetTrackAlternate`
- `GetTrackAlternate`
- `SetMediaLanguage`
- `GetMediaLanguage`
- `SetMediaQuality`
- `GetMediaQuality`

For further information about alternate tracks, see *QuickTime Movie Internals Guide*.

Editing Tracks

The Movie Toolbox provides a number of functions that allow your application to perform editing operations on tracks. These functions work with track segments (pieces of a track that are defined by a starting time and duration) and therefore give you a great deal of control over the editing process. These functions are similar to the low-level editing functions for movies that were described earlier in this chapter. However, these functions may copy movie data, if required by the operation.

When you edit a track you may change the duration of the movie that contains that track.

The `CopyTrackSettings` function lets you copy certain important settings from one track to another.

You can use the `InsertTrackSegment` function to copy a segment from one track to another, by reference or by moving data, or to copy a segment within a track. The `InsertTrackEmptySegment` function allows you to insert an empty segment into a track.

You can use the `InsertMediaIntoTrack` function to insert a media into a track.

Your application can delete a segment from a track by calling the `DeleteTrackSegment` function.

You can change a segment's duration by calling the `ScaleTrackSegment` function. This function stretches or shrinks the segment to accommodate a specified duration.

You can use the `GetTrackEditRate` function to determine the rate of the track edit of a specified track at an indicated time.

Working With Media Samples

This chapter describes the functions your application can use to determine information about a movie's sample data, such as the size in bytes of the data in a movie, track, or media, or the number of samples in a media, or the media's sample description.

- `GetMovieDataSize`
- `GetTrackDataSize`
- `GetMediaDataSize`
- `GetMediaSampleCount`
- `GetMediaSampleDescriptionCount`
- `GetMediaSampleDescription`
- `SetMediaSampleDescription`
- `MediaTimeToSampleNum`
- `SampleNumToMediaTime`

The Movie Toolbox provides a number of functions that allow applications to determine information about a movie's sample data. This section discusses these functions. See Adding Samples to Media Structures (page 35) for information about functions that allow you to retrieve sample data from a media.

Your application can use the `GetMovieDataSizeMovie`, `GetTrackDataSizeMovie`, and `GetMediaDataSize` functions to determine the size, in bytes, of the data stored in a media, movie, or track.

You can use the `GetMediaSampleDescriptionCount` and `GetMediaSampleDescription` functions to retrieve a media's sample descriptions. The `SetMediaSampleDescription` function enables you to change the contents of a particular sample description associated with a media. The `GetMediaSampleCount` function determines the number of samples in a media. The `SampleNumToMediaTime` and `MediaTimeToSampleNum` functions allow you to convert from a time value to a sample number and vice versa. You can use the functions described in Finding Interesting Times (page 36) to locate specific samples in a media.

Adding Samples to Media Structures

This section describes several functions your application can use to directly manipulate media samples. Note that `GetMediaSampleReferences` and `AddMediaSampleReferences` are plural forms added to `GetMediaSampleReference` and `AddMediaSampleReference`. They let you work with multiple samples at once, which is generally more efficient.

- `BeginMediaEdits`
- `EndMediaEdits`
- `AddMediaSample`

- `AddMediaSampleReference`
- `GetMediaSample`
- `GetMediaSampleReference`
- `GetMediaSampleReferences`
- `AddMediaSampleReferences`
- `SetMediaDefaultDataRefIndex`
- `SetMediaPreferredChunkSize`
- `GetMediaPreferredChunkSize`

This section describes Movie Toolbox functions that directly manipulate media samples. These functions are used only by applications that create movies or add data to existing movies.

You add samples to a media by calling the `AddMediaSample` function. You can indicate that the sample to be added is not a sync sample. **Sync samples** do not rely on preceding frames for content. Some compression algorithms conserve space by eliminating duplication between consecutive frames in a sample. In image data, sync samples are referred to as *key frames.*

You can obtain the data in a media sample by calling the `GetMediaSample` function. If you are going to add samples to a media, you must do so within a media-editing session. You start a media-editing session by calling the `BeginMediaEdits` function. Once you have finished adding samples to the media, you end the editing session by calling the `EndMediaEdits` function.

Once you have added samples to a media, you can work with references to those samples by calling the `AddMediaSampleReference` and `GetMediaSampleReference` functions. You do not have to be in a media-editing session to use these functions.

Finding Interesting Times

The Movie Toolbox provides a set of functions that help you locate samples in movies, tracks, and media structures. These functions are based on the concept of "interesting times." An interesting time refers to a time value in a movie, track, or media that meets certain search criteria. You specify the search criteria to the Movie Toolbox. The Movie Toolbox then scans the movie, track, or media, and locates time values that meet those search criteria.

You can use these functions to search through image sequences. For example, you may want to locate each frame in an image sequence. Or you may be more interested in key frames, especially if you are trying to optimize display performance. In image data, sync samples are referred to as **key frames.** An easy way to determine whether a movie has been edited is to look for track edits in the movie data. You may also be interested in searching for samples in a movie's media. If you set the appropriate search criteria, the Movie Toolbox locates the appropriate frames for you. You need the functions described in this section because QuickTime doesn't have a fixed rate. Each frame can have its own duration.

The Movie Toolbox identifies an interesting time by specifying its starting time and duration. The starting time indicates the time in the movie, track, or media where the search criteria are met. The duration indicates the length of time during which the search criteria remain in effect. For example, if you are looking for samples in a media, the start time would indicate the beginning of the sample, and the duration would indicate the

length of time to the next sample. In this case, you could find the next media sample by adding the duration to the start time. These duration values are always positive; you determine the direction of the search by setting the sign of the rate value you supply to the functions.

Note that movie interesting times are defined in the scope of the movie as a whole. As a result, one interesting time ends when another interesting time starts in any track in the movie. For example, if you are looking for key frames in a movie, the duration value from one interesting time tells you when the next key frame starts. However, that second key frame may be in a different track in the movie. Therefore, the duration of the interesting time does not necessarily correspond to the duration of the key frame.

You can use the `GetMovieNextInterestingTime` function to locate times of interest in a movie. The `GetTrackNextInterestingTime` function lets you work with tracks. Use the `GetMediaNextInterestingTime` function to locate samples in a media.

Working With Movie User Data

Each movie, track, and media can contain a user data list, which your application can use in any way you want. A **user data list** contains all the user data for a movie, track, or media. Each user data list may contain one or more **user data items.**

The functions your application can use to work with movie user data are listed below. Each movie, track and media can contain a user data list, which can be used for any purpose you like.

- `GetMovieUserData`
- `GetTrackUserData`
- `GetMediaUserData`
- `GetNextUserDataType`
- `CountUserDataType`
- `AddUserData`
- `GetUserData`
- `RemoveUserData`
- `AddUserDataText`
- `GetUserDataText`
- `RemoveUserDataText`
- `SetUserDataItem`
- `GetUserDataItem`
- `NewUserData`
- `DisposeUserData`
- `PutUserDataIntoHandle`
- `NewUserDataFromHandle`

Each user data item carries a type identifier. This type is stored in a long integer. Apple has reserved all lowercase user data type values. You are free to create user data type values using uppercase letters. Apple recommends using type values that begin with the copyright symbol.

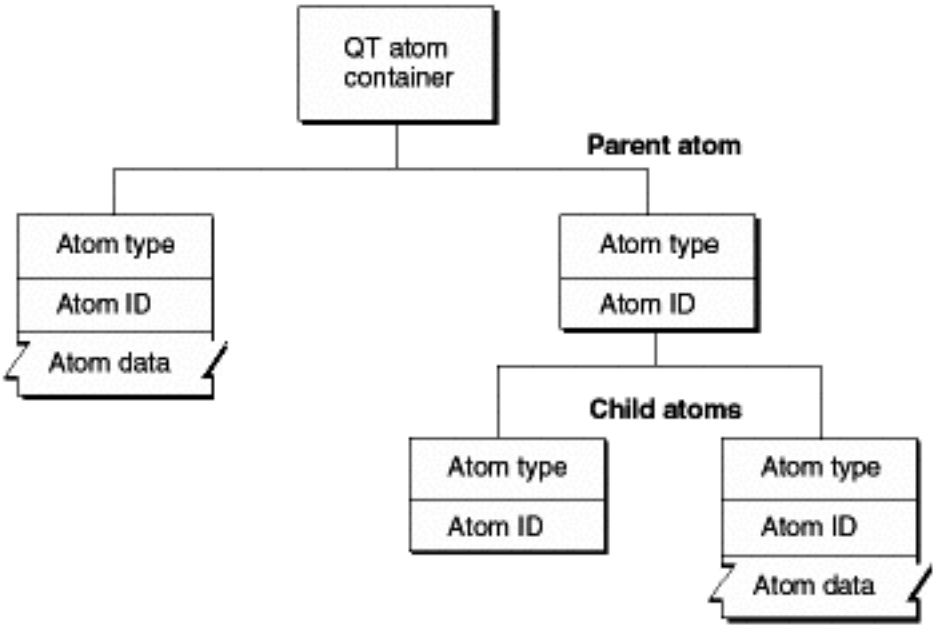
Understanding QuickTime Atoms

This chapter discusses QuickTime atoms, a basic structure for storing information in QuickTime. It also describes the functions used to create, dispose of, read from, and store to QuickTime atom containers. The QT atom is an enhancement of the existing atom data structure. Most QuickTime data structures (movies, tracks, media) are built of atoms. Newer QuickTime data structures (timecode tracks, sprites) are implemented using QT atoms.

Atom Structures and IDs

A QT **atom container** is a basic memory structure for storing information in QuickTime. You can use a QT atom container to construct arbitrarily complex hierarchical data structures. You can think of a newly-created QT atom container as the root of a tree structure that contains no children. A QT atom container contains **QT atoms** (Figure 7-1). Each QT atom contains either data or other atoms. If a QT atom contains other atoms, it is a **parent atom** and the atoms it contains are its **child atoms.** If a QT atom contains data, it is called a **leaf atom.**

Figure 7-1 QT atom container with parent and child atoms



Each QT atom has an offset that describes the atom's position within the QT atom container. In addition, each QT atom has a type and an ID. The atom type describes the kind of information the atom represents. The atom ID is used to differentiate child atoms of the same type with the same parent; an atom's ID must be unique for a given parent and type. In addition to the atom ID, each atom has a 1-based index that describes its order relative to other child atoms of the same parent. You can uniquely identify a QT atom in three ways:

- by its offset within its QT atom container
- by its parent atom, type, and index
- by its parent atom, type, and ID

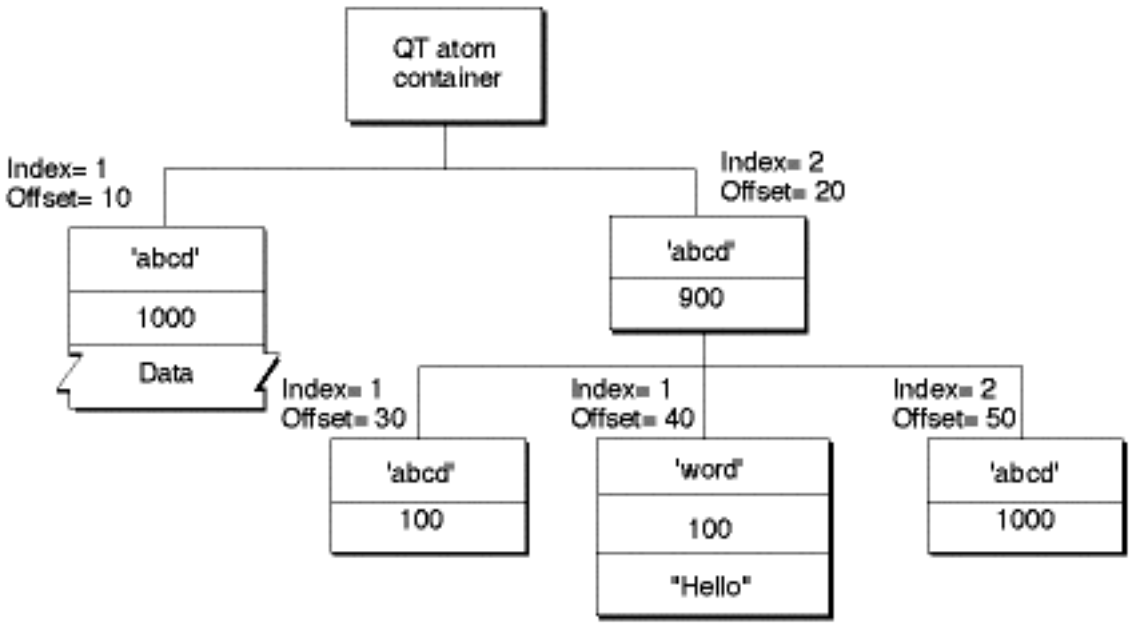
You can store and retrieve atoms in a QT atom container by index, ID, or both. For example, to use a QT atom container as a dynamic array or tree structure, you can store and retrieve atoms by index. To use a QT atom container as a database, you can store and retrieve atoms by ID. You can also create, store, and retrieve atoms using both ID and index to create an arbitrarily complex, extensible data structure.

Warning: Since QT atoms are offsets into a data structure, they can be changed during editing operations on QT atom containers, such as inserting or deleting atoms. For a given atom, editing child atoms is safe, but editing sibling or parent atoms invalidates that atom's offset.

Note: For cross-platform purposes, all data in a QT atom is expected to be in big-endian format.

Figure 7-2 shows a QT atom container that has two child atoms. The first child atom (offset = 10) is a leaf atom that has an atom type of `'abcd'`, an ID of 1000, and an index of 1. The second child atom (offset = 20) has an atom type of `'abcd'`, an ID of 900, and an index of 2. Because the two child atoms have the same type, they must have different IDs. The second child atom is also a parent atom of three atoms.

Figure 7-2 QT atom container example



The first child atom (offset = 30) has an atom type of `'abcd'`, an ID of 100, and an index of 1. It does not have any children, nor does it have data. The second child atom (offset = 40) has an atom type of `'word'`, an ID of 100, and an index of 1. The atom has data, so it is a leaf atom. The second atom (offset = 40) has the same ID as the first atom (offset = 30), but a different atom type. The third child atom (offset = 50) has an atom type of `'abcd'`, an ID of 1000, and an index of 2. Its atom type and ID are the same as that of another atom (offset = 20) with a different parent.

As a developer, you do not need to parse QT atoms yourself. Instead, you can use the QT atom functions to create atom containers, add atoms to and remove atoms from atom containers, search for atoms in atom containers, and retrieve data from atoms in atom containers.

Most QT atom functions take two parameters to specify a particular atom: the atom container that contains the atom and the offset of the atom in the atom container data structure. You obtain an atom's offset by calling either `QTFindChildByID` or `QTFindChildByIndex`. An atom's offset may be invalidated if the QT atom container that contains it is modified.

When calling any QT atom function for which you specify a parent atom as a parameter, you can pass the constant `kParentAtomIsContainer` as an atom offset to indicate that the specified parent atom is the atom container itself. For example, you would call the `QTFindChildByIndex` function and pass `kParentAtomIsContainer` constant for the parent atom parameter to indicate that the requested child atom is a child of the atom container itself.

Creating and Disposing of Atom Containers

Before you can add atoms to an atom container, you must first create the container by calling `QTNewAtomContainer`. The code sample shown in Listing 7-1 calls `QTNewAtomContainer` to create an atom container.

Listing 7-1 Creating a new atom container

```
QTAtomContainer spriteData;
OSErr err
// create an atom container to hold a sprite's data
err=QTNewAtomContainer (&spriteData);
```

When you have finished using an atom container, you should dispose of it by calling the `QTDisposeAtomContainer` function. The sample code shown in Listing 7-2 calls `QTDisposeAtomContainer` to dispose of the `spriteData` atom container.

Listing 7-2 Disposing of an atom container

```
if (spriteData)
    QTDisposeAtomContainer (spriteData);
```

Creating New Atoms

You can use the `QTInsertChild` function to create new atoms and insert them in a QT atom container. The `QTInsertChild` function creates a new child atom for a parent atom. The caller specifies an atom type and atom ID for the new atom. If you specify a value of 0 for the atom ID, `QTInsertChild` assigns a unique ID to the atom.

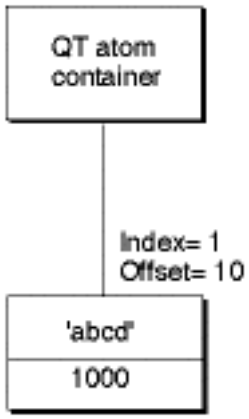
`QTInsertChild` inserts the atom in the parent's child list at the index specified by the `index` parameter; any existing atoms at the same index or greater are moved toward the end of the child list. If you specify a value of 0 for the `index` parameter, `QTInsertChild` inserts the atom at the end of the child list.

The code sample in Listing 7-3 creates a new QT atom container and calls `QTInsertChild` to add an atom. The resulting QT atom container is shown in Figure 7-3. The offset value 10 is returned in the `firstAtom` parameter.

Listing 7-3 Creating a new QT atom container and calling QTInsertChild to add an atom

```
QTAtom firstAtom;
QTAtomContainer container;
OSErr err
err = QTNewAtomContainer (&container);
if (!err)
    err = QTInsertChild (container, kParentAtomIsContainer, 'abcd',
        1000, 1, 0, nil, &firstAtom);
```

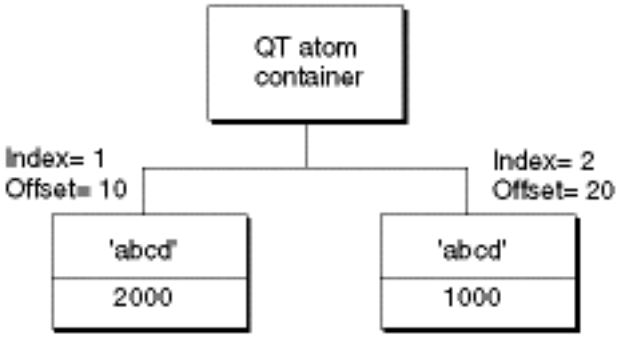
Figure 7-3 QT atom container after inserting an atom



The following code sample calls `QTInsertChild` to create a second child atom. Because a value of 1 is specified for the `index` parameter, the second atom is inserted in front of the first atom in the child list; the index of the first atom is changed to 2. The resulting QT atom container is shown in Figure 7-4.

```
QTAtom secondAtom;
FailOSErr (QTInsertChild (container, kParentAtomIsContainer, 'abcd',
    2000, 1, 0, nil, &secondAtom));
```

Figure 7-4 QT atom container after inserting a second atom



You can call the `QTFindChildByID` function to retrieve the changed offset of the first atom that was inserted, as shown in the following example. In this example, the `QTFindChildByID` function returns an offset of 20.

```
firstAtom = QTFindChildByID (container, kParentAtomIsContainer, 'abcd',
    1000, nil);
```

Listing 7-4 shows how the `QTInsertChild` function inserts a leaf atom into the atom container `sprite`. The new leaf atom contains a sprite image index as its data.

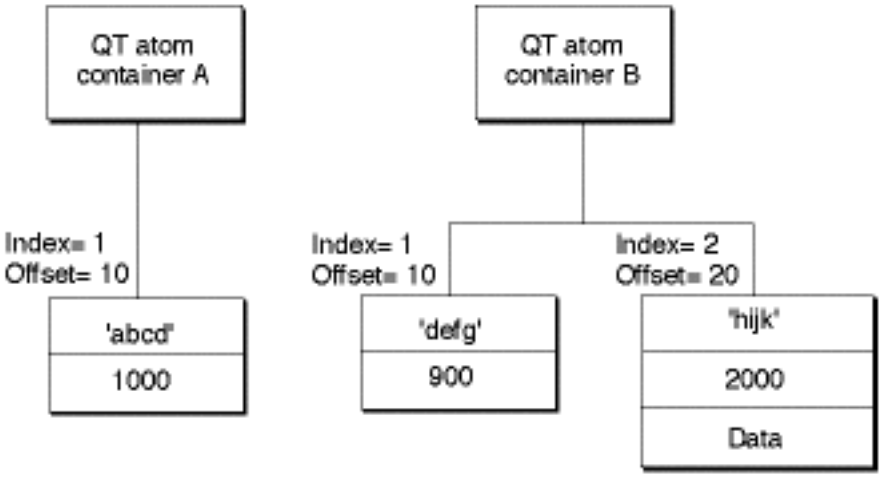
Listing 7-4 Inserting a child atom

```
if ((propertyAtom = QTFindChildByIndex (sprite, kParentAtomIsContainer,
    kSpritePropertyImageIndex, 1, nil)) == 0)
    FailOSErr (QTInsertChild (sprite, kParentAtomIsContainer,
        kSpritePropertyImageIndex, 1, 1, sizeof(short),&imageIndex,
        nil));
```

Copying Existing Atoms

QuickTime provides several functions for copying existing atoms within an atom container. The `QTInsertChildren` function inserts a container of atoms as children of a parent atom in another atom container. Figure 7-5 shows two example QT atom containers, A and B.

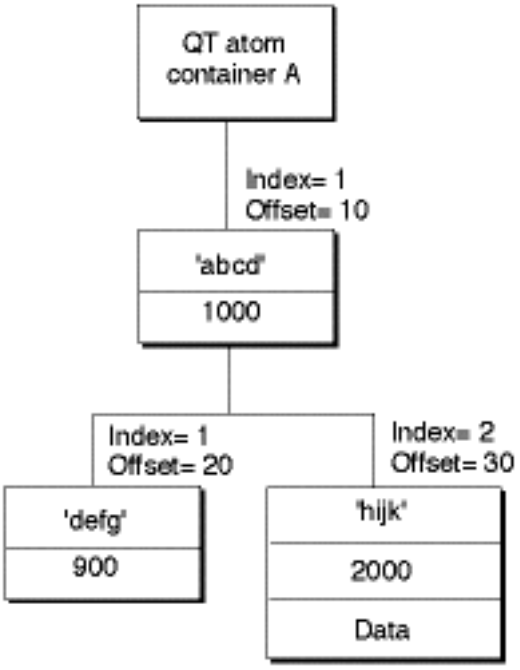
Figure 7-5 Two QT atom containers, A and B



The following code sample calls `QTFindChildByID` to retrieve the offset of the atom in container A. Then, the code sample calls the `QTInsertChildren` function to insert the atoms in container B as children of the atom in container A. Figure 7-6 shows what container A looks like after the atoms from container B have been inserted.

```
QTAtom targetAtom;
targetAtom = QTFindChildByID (containerA, kParentAtomIsContainer, 'abcd',
    1000, nil);
FailOSErr (QTInsertChildren (containerA, targetAtom, containerB));
```

Figure 7-6 QT atom container after child atoms have been inserted



In Listing 7-5, the `QTInsertChild` function inserts a parent atom into the atom container `theSample`. Then, the code calls `QTInsertChildren` to insert the container `theSprite` into the container `theSample`. The parent atom is `newSpriteAtomMovie` Data Types.

Listing 7-5 Inserting a container into another container

```
FailOSErr (QTInsertChild (theSample, kParentAtomIsContainer,
    kSpriteAtomType, spriteID, 0, 0, nil, &newSpriteAtom));
FailOSErr (QTInsertChildren (theSample, newSpriteAtom, theSprite));
```

QuickTime provides three other functions you can use to manipulate atoms in an atom container. The `QTReplaceAtom` function replaces an atom and its children with a different atom and its children. You can call the `QTSwapAtoms` function to swap the contents of two atoms in an atom container; after swapping, the ID and index of each atom remains the same. The `QTCopyAtom` function copies an atom and its children to a new atom container.

Retrieving Atoms From an Atom Container

QuickTime provides functions you can use to retrieve information about the types of a parent atom's children, to search for a specific atom, and to retrieve a leaf atom's data.

You can use the `QTCountChildrenOfType` and `QTGetNextChildType` functions to retrieve information about the types of an atom's children. The `QTCountChildrenOfType` function returns the number of children of a given atom type for a parent atom. The `QTGetNextChildType` function returns the next atom type in the child list of a parent atom.

You can use the `QTFindChildByIndex`, `QTFindChildByID`, and `QTNextChildAnyType` functions to retrieve an atom. You call the `QTFindChildByIndex` function to search for and retrieve a parent atom's child by its type and index within that type.

Listing 7-6 shows the sample code function `SetSpriteData`, which updates an atom container that describes a sprite. For each property of the sprite that needs to be updated, `SetSpriteData` calls `QTFindChildByIndex` to retrieve the appropriate atom from the atom container. If the atom is found, `SetSpriteData` calls `QTSetAtomData` to replace the atom's data with the new value of the property. If the atom is not found, `SetSpriteData` calls `QTInsertChild` to add a new atom for the property.

Listing 7-6 Finding a child atom by index

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

    // if the sprite's visible property has a new value
    if (visible)
    {
        // retrieve the atom for the visible property --
        // if none exists, insert one
        if ((propertyAtom = QTFindChildByIndex (sprite,
            kParentAtomIsContainer, kSpritePropertyVisible, 1,
```

```
            nil)) == 0)
            FailOSErr (QTInsertChild (sprite, kParentAtomIsContainer,
                kSpritePropertyVisible, 1, 1, sizeof(short), visible,
                nil))

        // if an atom does exist, update its data
        else
            FailOSErr (QTSetAtomData (sprite, propertyAtom,
                sizeof(short), visible));
    }

    // ...
    // handle other sprite properties
    // ...
}
```

You can call the `QTFindChildByID` function to search for and retrieve a parent atom's child by its type and ID. The sample code function `AddSpriteToSample`, shown in Listing 7-7, adds a sprite, represented by an atom container, to a key sample, represented by another atom container. `AddSpriteToSample` calls `QTFindChildByID` to determine whether the atom container `theSample` contains an atom of type `kSpriteAtomType` with the ID `spriteIDMovie` Data Types. If not, `AddSpriteToSample` calls `QTInsertChild` to insert an atom with that type and ID. A value of 0 is passed for the `index` parameter to indicate that the atom should be inserted at the end of the child list. A value of 0 is passed for the `dataSize` parameter to indicate that the atom does not have any data. Then, `AddSpriteToSample` calls `QTInsertChildren` to insert the atoms in the container `theSprite` as children of the new atom. `FailIf` and `FailOSErr` are macros that exit the current function when an error occurs.

Listing 7-7 Finding a child atom by ID

```
OSErr AddSpriteToSample (QTAtomContainer theSample,
    QTAtomContainer theSprite, short spriteID)
{
    OSErr err = noErr;
    QTAtom newSpriteAtom;
    FailIf (QTFindChildByID (theSample, kParentAtomIsContainer,
        kSpriteAtomType, spriteID, nil), paramErr);
    FailOSErr (QTInsertChild (theSample, kParentAtomIsContainer,
        kSpriteAtomType, spriteID, 0, 0, nil, &newSpriteAtom));
    FailOSErr (QTInsertChildren (theSample, newSpriteAtom, theSprite));
}
```

Once you have retrieved a child atom, you can call `QTNextChildAnyType` function to retrieve subsequent children of a parent atom. `QTNextChildAnyType` returns an offset to the next atom of any type in a parent atom's child list. This function is useful for iterating through a parent atom's children quickly.

QuickTime also provides functions for retrieving an atom's type, ID, and data. You can call `QTGetAtomTypeAndID` function to retrieve an atom's type and ID. You can access an atom's data in one of three ways.

- To copy an atom's data to a handle, you can use the `QTCopyAtomDataToHandle` function.
- To copy an atom's data to a pointer, you can use the `QTCopyAtomDataToPtr` function.
- To access an atom's data directly, you should lock the atom container in memory by calling `QTLockContainerMovie` Data Types. Once the container is locked, you can call `QTGetAtomDataPtr` to retrieve a pointer to an atom's data. When you have finished accessing the atom's data, you should call the `QTUnlockContainer` function to unlock the container in memory.

Modifying Atoms

QuickTime provides functions that you can call to modify attributes or data associated with an atom in an atom container. To modify an atom's ID, you call the function `QTSetAtomIDMovie` Data Types.

You use the `QTSetAtomData` function to update the data associated with a leaf atom in an atom container. The `QTSetAtomData` function replaces a leaf atom's data with new data. The code sample in Listing 7-8 calls `QTFindChildByIndex` to determine whether an atom container contains a sprite's visible property. If so, the sample calls `QTSetAtomData` to replace the atom's data with a new visible property.

Listing 7-8 Modifying an atom's data

```
QTAtom propertyAtom;
// if the atom isn't in the container, add it
if ((propertyAtom = QTFindChildByIndex (sprite, kParentAtomIsContainer,
    kSpritePropertyVisible, 1, nil)) == 0)
    FailOSErr (QTInsertChild (sprite, kParentAtomIsContainer,
        kSpritePropertyVisible, 1, 0, sizeof(short), visible, nil))
// if the atom is in the container, replace its data
else
    FailOSErr (QTSetAtomData (sprite, propertyAtom, sizeof(short),
        visible));
```

Removing Atoms From an Atom Container

To remove atoms from an atom container, you can use the `QTRemoveAtom` and `QTRemoveChildren` functions. The `QTRemoveAtom` function removes an atom and its children, if any, from a container. The `QTRemoveChildren` function removes an atom's children from a container, but does not remove the atom itself. You can also use `QTRemoveChildren` to remove all the atoms in an atom container. To do so, you should pass the constant `kParentAtomIsContainer` for the `atom` parameter.

The code sample shown in Listing 7-9 adds override samples to a sprite track to animate the sprites in the sprite track. The `sample` and `spriteData` variables are atom containers. The `spriteData` atom container contains atoms that describe a single sprite. The `sample` atom container contains atoms that describes an override sample.

Each iteration of the `for` loop calls `QTRemoveChildren` to remove all atoms from both the `sample` and the `spriteData` containers. The sample code updates the index of the image to be used for the sprite and the sprite's location and calls `SetSpriteData`, which adds the appropriate atoms to the `spriteData` atom container. Then, the sample code calls `AddSpriteToSample` to add the `spriteData` atom container to the `sample` atom container. Finally, when all the sprites have been updated, the sample code calls `AddSpriteSampleToMedia` to add the override sample to the sprite track.

Listing 7-9 Removing atoms from a container

```
QTAtomContainer sample, spriteData;
// ...
// add the sprite key sample
// ...
// add override samples to make the sprites spin and move
for (i = 1; i <= kNumOverrideSamples; i++)
```

```
{
    QTRemoveChildren (sample, kParentAtomIsContainer);
    QTRemoveChildren (spriteData, kParentAtomIsContainer);

    // ...
    // update the sprite:
    // - update the imageIndex
    // - update the location
    // ...
    // add atoms to spriteData atom container
    SetSpriteData (spriteData, &location, nil, nil, &imageIndex);
    // add the spriteData atom container to sample
    err = AddSpriteToSample (sample, spriteData, 2);
    // ...
    // update other sprites
    // ...
    // add the sample to the media
    err = AddSpriteSampleToMedia (newMedia, sample,
        kSpriteMediaFrameDuration, false);
}
```

Creating and Modifying QT Atom Containers

The following functions can be used to create and modify QT atom containers:

- `QTNewAtomContainer`
- `QTInsertChild`
- `QTInsertChildren`
- `QTReplaceAtom`
- `QTSwapAtoms`
- `QTSetAtomID`
- `QTSetAtomData`
- `QTCopyAtom`
- `QTLockContainer`
- `QTGetAtomDataPtr`
- `QTUnlockContainer`
- `QTRemoveAtom`
- `QTRemoveChildren`
- `QTDisposeAtomContainer`

Retrieving Atoms and Atom Data

This following functions can be used to retrieve QT Atoms and atom data. Each QT atom contains either data or other atoms.

- `QTGetNextChildType`
- `QTCountChildrenOfType`
- `QTFindChildByIndex`
- `QTFindChildByID`
- `QTNextChildAnyType`
- `QTCopyAtomDataToHandle`
- `QTCopyAtomDataToPtr`
- `QTGetAtomTypeAndID`

Constants for QT Atom Functions

You can pass the `kParentAtomIsContainer` constant to QT atom functions that take an atom container and a parent atom as parameters. When passed in place of the parent atom, this constant indicates that the parent atom is the atom container itself.

```
enum {
    kParentAtomIsContainer  = 0
};
```

QT Atom

The `QTAtom` data type represents the offset of an atom within an atom container.

```
typedef long QTAtom;
```

QT Atom Type and ID

The `QTAtomType` data type represents the type of a QT atom. To be valid, a QT atom's type must have a nonzero value.

```
typedef long QTAtomType;
```

The `QTAtomID` data type represents the ID of a QT atom. To be valid, a QT atom's ID must have a nonzero value.

```
typedef long QTAtomID;
```

QT Atom Container

The `QTAtomContainer` data type is a handle to a QT atom container. Your application never modifies the contents of a QT atom container directly. Instead, you use the functions provided by QuickTime for creating and manipulating QT atom containers.

```
typedef Handle QTAtomContainer;
```

Timecode Media Handler Functions

QuickTime includes support for timecode tracks. Timecode tracks allow you to store external timecode information, such as SMPTE timecodes, in your QuickTime movies. QuickTime provides a timecode media handler that interprets the data in these tracks. This chapter discusses the functions and structures that allow you to use the timecode media handler.

About Timecodes

The timecode media handler allows QuickTime movies to store timing information that is derived from the movie's original source material. Every QuickTime movie contains QuickTime-specific timing information, such as frame duration. This information affects how QuickTime interprets and plays the movie.

The timecode media handler allows QuickTime movies to store additional timing information that is not created by or for QuickTime. This additional timing information would typically be derived from the original source material; for example, as a SMPTE timecode. In essence, you can think of the timecode media handler as providing a link between the digital QuickTime-specific timing information and the original analog timing information from the source material.

A movie's timecode is stored in a timecode track. Timecode tracks contain

- source identification information (this identifies the source; for example, a given videotape)
- timecode format information (this specifies the characteristics of the timecode and how to interpret the timecode information)
- frame numbers (these allow QuickTime to map from a given movie time, in terms of QuickTime time values, to its corresponding timecode value)

Apple Computer has defined the information that is stored in the track in a manner that is independent of any specific timecode standard. The format of this information is sufficiently flexible to accommodate all known timecode standards, including SMPTE timecoding. The timecode format information provides QuickTime the parameters for understanding the timecode and converting QuickTime time values into timecode time values (and vice versa).

One key timecode attribute relates to the technique used to synchronize timecode values with video frames. Most video source material is recorded at whole-number frame rates. For example, both PAL and SECAM video contain exactly 25 frames per second. However, some video source material is not recorded at whole-number frame rates. In particular, NTSC color video contains 29.97 frames per second (though it is typically referred to as 30 frames-per-second video). However, NTSC timecode values correspond to the full 30 frames-per-second rate; this is a holdover from NTSC black-and-white video. For such video sources, you need a mechanism that corrects the error that will develop over time between timecode values and actual video frames.

A common method for maintaining synchronization between timecode values and video data is called dropframe. Contrary to its name, the dropframe technique actually skips timecode values at a predetermined rate in order to keep the timecode and video data synchronized. It does not actually drop video frames. In NTSC color video, which uses the dropframe technique, the timecode values skip two frame values every minute, except for minute values that are evenly divisible by ten. So NTSC timecode values, which are expressed as HH:MM:SS:FF (hours, minutes, seconds, frames) skip from 00:00:59:29 to 00:01:00:02 (skipping 00:01:00:00 and 00:01:00:01). There is a flag in the timecode definition structure that indicates whether the timecode uses the dropframe technique.

You can make the toolbox display the timecode when a movie is played. Use the `TCSetTimeCodeFlags` function to turn the timecode display on and off. Note that the timecode track must be enabled for this display to work.

You store the timecode's source identification information in a user data item. Create a user data item with a type value of `TCSourceRefNameType`. Store the source information as a text string. This information might contain the name of the videotape from which the movie was created, for example.

The timecode media handler provides functions that allow you to manipulate the source identification information. The following sample code demonstrates one way to set the source tape name in a timecode media's sample description.

```
void setTimeCodeSourceName (Media timeCodeMedia,
                            TimeCodeDescriptionHandle tcdH,
                            Str255 tapeName, ScriptCode tapeNameScript)
{
    UserData srcRef;
    if (NewUserData(&srcRef) == noErr) {
        Handle nameHandle;
        if (PtrToHand(&tapeName[1], &nameHandle, tapeName[0]) == noErr) {
            if (AddUserDataText (srcRef, nameHandle,'name', 1,
                                 tapeNameScript) == noErr) {
                TCSetSourceRef (GetMediaHandler (timeCodeMedia),
                                 tcdH,
                                 srcRef);
            }
            DisposeHandle(nameHandle);
        }
        DisposeUserData(srcRef);
    }
}
```

Creating a Timecode Track

You can create a timecode track and media in the same manner that you create any other track. Call the `NewMovieTrack` function to create the timecode track, and use the `NewTrackMedia` function to create the track's media. Be sure to specify a media type value of `TimeCodeMediaType` when you call the `NewTrackMedia` function.

You can define the relationship between a timecode track and one or more movie tracks using the toolbox's new track reference functions. You then proceed to add samples to the track, as appropriate.

Each sample in the timecode track provides timecode information for a span of movie time. The sample includes duration information. As a result, you typically add each timecode sample after you have created the corresponding content track or tracks.

The timecode media sample description contains the control information that allows QuickTime to interpret the samples. This includes the timecode format information. The actual sample data contains a frame number that identifies one or more content frames that use this timecode. Stored as a `long`, this value identifies the first frame in the group of frames that use this timecode. In the case of a movie made from source material that contains no edits, you would only need one sample. When the source material contains edits, you typically need one sample for each edit, so that QuickTime can resynchronize the timecode information with the movie. Those samples contain the frame numbers of the frames that begin each new group of frames.

The timecode description structure defines the format and content of a timecode media sample description, as follows:

```
typedef struct TimeCodeDescription {
    long                descSize;           /* size of the structure */
    long                dataFormat;         /* sample type */
    long                resvd1;             /* reserved; set to 0 */
    short               resvd2;             /* reserved; set to 0 */
    short               dataRefIndex;       /* data reference index */
    long                flags;              /* reserved; set to 0 */
    TimeCodeDef         timeCodeDef;        /* timecode format information */
    long                srcRef[1];          /* source information */
} TimeCodeDescription, *TimeCodeDescriptionPtr, **TimeCodeDescriptionHandle;
```

Term	Definition
`descSize`	Specifies the size of the sample description, in bytes.
`dataFormat`	Indicates the sample description type (`TimeCodeMediaTypeMovie` Data Types).
`resvd1`	Reserved for use by Apple. Set this field to 0.
`resvd2`	Reserved for use by Apple. Set this field to 0.
`dataRefIndex`	Contains an index value indicating which of the media's data references contains the sample data for this sample description.
`flags`	Reserved for use by Apple. Set this field to 0.
`timeCodeDef`	Contains a timecode definition structure that defines timecode format information.
`srcRef`	Contains the timecode's source information. This is formatted as a user data item that is stored in the sample description. The media handler provides functions that allow you to get and set this data.

The timecode definition structure contains the timecode format information. This structure is defined as follows:

```
typedef struct TimeCodeDef {
    long            flags;              /* timecode control flags */
    TimeScale       fTimeScale;         /* timecode's time scale */
    TimeValue       frameDuration;      /* how long each frame lasts */
    unsigned char   numFrames;          /* number of frames per second */
```

```
} TimeCodeDef;
```

Parameter	Definition
flags	Contains flags that provide some timecode format information. The following flags are defined:

Flag	Definition
tcDropFrame	Indicates that the timecode drops frames occasionally in order to stay in synchronization. Some timecodes run at other than a whole number of frames per second. For example, NTSC video runs at 29.97 frames per second. In order to resynchronize between the timecode rate and a 30 frames-per-second playback rate, the timecode drops a frame at a predictable time (in much the same way that leap years keep the calendar synchronized). Set this flag to 1 if the timecode uses the dropframe technique.
tc24HourMax	Indicates that the timecode values wrap at 24 hours. Set this flag to 1 if the timecode hour value wraps (that is, returns to 0) at 24 hours.
tcNegTimesOK	Indicates that the timecode supports negative time values. Set this flag to 1 if the timecode allows negative values.
tcCounter	Indicates that the timecode should be interpreted as a simple counter, rather than as a time value. This allows the timecode to contain either time information or counter (such as a tape counter) information. Set this flag to 1 if the timecode contains counter information.
fTimeScale	Contains the time scale for interpreting the frameDuration field. This field indicates the number of time units per second.
frameDuration	Specifies how long each frame lasts, in the units defined by the fTimeScale field.
numFrames	Indicates the number of frames stored per second. In the case of timecodes that are interpreted as counters, this field indicates the number of frames stored per timer tick.

The best way to understand how to format and interpret the timecode definition structure is to consider an example. If you were creating a movie from an NTSC video source recorded at 29.97 frames per second, using SMPTE timecodes, you would format the timecode definition structure as follows:

```
TimeCodeDef.flags = tcDropFrame | tc24HourMax;
TimeCodeDef.fTimeScale = 2997;          /* units */
TimeCodeDef.frameDuration = 100;        /* relates units to frames */
TimeCodeDef.numFrames = 30;             /* whole frames per second */
```

The movie's natural frame rate of 29.97 frames per second is obtained by dividing the fTimeScale value by the frameDuration (2997 / 100). Note that the flags field indicates that the timecode uses the dropframe technique to resync the movie's natural frame rate of 29.97 frames per second with its playback rate of 30 frames per second.

Given a timecode definition, you can freely convert from frame numbers to time values and from time values to frame numbers. For a time value of 00:00:12:15 (HH:MM:SS:FF), you would obtain a frame number of 375 ((12*30) + 15). The timecode media handler provides a number of functions that allow you to perform these conversions.

When you use the timecode media handler to work with time values, the media handler uses timecode records to store the time values. The timecode record allows you to interpret the time information as either a time value (HH:MM:SS:FF) or a counter value. The timecode record is defined as follows:

```
typedef union TimeCodeRecord {
    TimeCodeTime        t;          /* value interpreted as time */
    TimeCodeCounter     c;          /* value interpreted as counter */
} TimeCodeRecord;

typedef struct TimeCodeTime {
    unsigned char       hours;      /* time: hours */
    unsigned char       minutes;    /* time: minutes */
    unsigned char       seconds;    /* time: seconds */
    unsigned char       frames;     /* time: frames */
} TimeCodeTime;

typedef struct TimeCodeCounter {
    long                counter;    /* counter value */
} TimeCodeCounter;
```

When you are working with timecodes that allow negative time values, the `minutes` field of the `TimeCodeTime` structure (`TimeCodeRecord`.`t`.`minutesMovie` Data Types) indicates whether the time value is positive or negative. If the `tctNegFlag` bit of the `minutes` field is set to 1, the time value is negative.

Locating Tracks, Saving Movies, and Modifying Movie Properties

The Movie Toolbox provides a set of functions that help your application locate a movie's tracks and media structures. This chapter describes these functions and how to use them. In addition, the chapter discusses functions you can use to save movies, capture and restore the edit state of a movie, and modify movie properties. A later section of the chapter discusses support for progressive downloads.

The Movie Toolbox identifies a movie's tracks in two ways. First, every track in a movie has a unique ID value. This ID value is unique throughout the life of a movie, even after it has been saved. That is, no two tracks of a movie ever have the same ID, and no ID value is ever reused. Second, a movie's current tracks may be identified by their index value. Index values always range from 1 to the number of tracks in the movie. Track indexes provide a convenient way to access each track of a movie.

There are several functions that allow you to find a movie's tracks. You can use the `GetMovieTrackCount` function to determine the number of tracks in a movie. Use the `GetMovieTrack` function to obtain the track identifier for a specific track, given its ID. The `GetMovieIndTrack` function lets you obtain a track's identifier, given its track index.

You can obtain a track's ID value given its track identifier by calling the `GetTrackID` function.

You can determine the movie that contains a track by calling the `GetTrackMovie` function.

The `GetTrackMedia` function enables you to find a track's media. Conversely, you can find the track that uses a media by calling the `GetMediaTrack` function.

Saving Movies

The Movie Toolbox provides a set of high-level functions for storing movies within files. These files have a file type of `'MooV'` and a resource type of `'moov'`. Your application can gain access to existing movies with either the `NewMovieFromFile` function or the `NewMovieFromDataFork` function. Once you have loaded the movie, your application uses the functions that are described in this section to save any changes you have made to the movie.

You can use the `AddMovieResource` function to add a new movie resource to a movie file. Your application can use this function to save a movie that it created using the functions. You can use the `UpdateMovieResource` function to replace an existing movie resource in a movie file. You can remove a movie resource by calling the `RemoveMovieResource` function.

The movie resources that your application creates with the `AddMovieResource` and `UpdateMovieResource` functions may contain references to movie data. These references identify the data that constitute the movie. However, the movie data can be stored outside of the movie file. If you want to create a movie file that contains all of its movie data, use the `FlattenMovie` or `FlattenMovieData` function. These functions can also be used to store the movie data in the movie file's data fork, or to interleave the media data to optimize performance.

The `PutMovieIntoHandle` function places a QuickTime movie into a handle. You can then convert the movie into specialized data formats.

The `HasMovieChanged` and `ClearMovieChanged` functions allow your application to work with the movie changed flag that is maintained by the Movie Toolbox. You can use this flag to determine whether a movie has been changed.

The movie changed flag indicates whether you have changed the movie. Such actions as editing the movie, adding samples to a media, or changing a data reference cause the flag to indicate that the movie has changed. There are several operations that the movie changed flag does not reflect, including changing the volume, rate, or time settings for the movie. These settings change frequently when a movie is played. Your application must monitor these settings itself.

The Movie Toolbox also supplies functions for storing and retrieving movies that are stored in the data fork of a file. These functions provide robust data reference resolution and improve low memory performance. The `NewMovieFromDataFork` function enables you to retrieve a movie that is stored anywhere in the data fork of a file. You can use the `PutMovieIntoDataFork` function to store an atom version of a specified movie in the data fork of a file.

Time Base Callback Functions

If your application uses QuickTime time bases, it may define callback functions that are associated with a specific time base. Your application can then use these callback functions to perform activities that are triggered by temporal events, such as a certain time being reached or a specified rate being achieved. The time base functions of the Movie Toolbox interact with clock components to schedule the invocation of these callback functions; clock components are responsible for invoking the callback function at its scheduled time. Your application can use the functions described in this section to establish your own callback function and to schedule callback events.

You can define three types of callback events. These types are distinguished by the nature of the temporal event that triggers the Movie Toolbox to call your function. The three types are

- events that are triggered at a specified time
- events that are triggered when the rate reaches a specified value
- events that are triggered when the time value of a time base changes by an amount different from the time base's rate

You specify a callback event's type when you define the callback event, using the `NewCallBack` function.

You specify whether your event can occur at interrupt time when you define the callback event, using the `NewCallBack` function. Your function is called closer to the triggering event at interrupt time, but it is subject to all the restrictions of interrupt functions (for example, your callback function cannot cause memory to be moved). If your function is not called at interrupt time, you are free of these restrictions; however, your function may be called later, because the invocation is delayed to avoid interrupt time.

The `NewCallBack` function allocates the memory to support a callback event. When you are done with the callback event, you dispose of it by calling the `DisposeCallBack` function.

You schedule a callback event by calling the `CallMeWhen` function. Call `CancelCallBack` function to unschedule a callback event.

You can retrieve the time base of a callback event by calling the `GetCallBackTimeBase` function. You can obtain the type of a callback event by calling the `GetCallBackType` function.

Creating and Disposing of Time Bases

This section discusses the Movie Toolbox functions your application can use to create and dispose of time bases.

The `NewTimeBase` function lets you create a new time base. You can use the `DisposeTimeBase` function to dispose of a time base once you are finished with it.

Time bases rely on either a clock component or another time base for their time source. You can use the `SetTimeBaseMasterTimeBase` function to cause one time base to be based on another time base. The `GetTimeBaseMasterTimeBase` allows you to determine the master time base of a given time base.

You can assign a clock component to a time base; that clock then acts as the master clock for the time base. You can use the `SetTimeBaseMasterClock` function to assign a clock component to a time base. The `GetTimeBaseMasterClock` function enables you to determine the clock component that is assigned to a time base. You can change the offset between a time base and its time source by calling the `SetTimeBaseZero` function.

You can set the time source of a movie by calling the `SetMovieMasterTimeBase` and `SetMovieMasterClock` functions.

Note: Although most time base functions can be used at interrupt time, several of the Movie Toolbox functions cannot. Consult the documentation for each function in the *QuickTime API Reference*.

Working with Movie Time

Every QuickTime movie has its own time base. A movie's time base allows all the tracks that make up the movie to be synchronized when the movie is played. The Movie Toolbox provides a number of functions that allow your application to determine and establish the time parameters of a movie. This section discusses those functions. Later sections in this chapter discuss the Movie Toolbox functions that allow you to work with the time parameters of tracks and media structures.

You can use the `GetMovieTimeBase` function to retrieve the time base for a movie.

You can work with a movie's current time by calling the `GetMovieTime`, `SetMovieTime`, and `SetMovieTimeValue` functions.

You can work with a movie's time scale by calling the `GetMovieTimeScale` and `SetMovieTimeScale` functions.

The Movie Toolbox can calculate the total duration of a movie. You can use the `GetMovieDuration` function to retrieve a movie's duration.

Your application can call the `GetMovieRate` and `SetMovieRate` to work with a movie's playback rate.

Working With Movie User Data

Each movie, track, and media can contain a user data list, which your application can use in any way you want. A **user data list** contains all the user data for a movie, track, or media. Each user data list may contain one or more **user data items.**

All QuickTime user data items share several attributes. First, each user data item carries a type identifier. This type is similar to a Resource Manager resource type, and is stored in a long integer. Apple has reserved all lowercase user data type values. You are free to create user data type values using uppercase letters. Apple recommends using type values that begin with the copyright symbol to specify user data items that store text data.

The Time Structure

The Movie Toolbox provides a number of functions that allow you to work with time specifications. Many of these functions require that you place a time specification in a data structure called a*time structure*. The time structure allows you to fully describe a time specification. The `TimeRecord` data type defines the format of a time structure.

```
struct TimeRecord
{
    CompTimeValue       value;      /* time value (duration or absolute) */
    TimeScale           scale;      /* units per second */
    TimeBase            base;       /* reference to the time base */
};
typedef struct TimeRecord TimeRecord;
```

Field	Description
`value`	Contains the time value. The time value defines either a duration or an absolute time by specifying the corresponding number of units of time. For durations, this is the number of time units in the period. For an absolute time, this is the number of time units since the beginning of the time coordinate system. The unit for this value is defined by the scale field. The time value is expressed as a `CompTimeValue` data type, which is a 64-bit integer quantity. This 64-bit quantity consists of two 32-bit integers, and it is defined by the `Int64` data type, which is described next in this section.
`scale`	Contains the time scale. This field specifies the number of units of time that pass each second. If you specify a value of 0, the time base uses its natural time scale.
`base`	Contains a reference to the time base. You obtain a time base by calling the Movie Toolbox's `GetMovieTimeBase` or `NewTimeBase` functions.

If the time structure defines a duration, set this field to `nil`. Otherwise, this field must refer to a valid time base.

You specify the time value in a time structure in a 64-bit integer value as follows:

```
typedef Int64 CompTimeValue;
```

The Movie Toolbox uses this format so that extremely large time values can be represented. The `Int64` data type defines the format of these signed 64-bit integers.

```
struct Int64
{
    long hi;    /* high-order 32 bits-value field in time structure */
    long lo;    /* low-order 32 bits-value field in time structure */
};
typedef struct Int64 Int64;
```

Field	Description
`hi`	Contains the high-order 32 bits of the value. The high-order bit represents the sign of the 64-bit integer.
`lo`	Contains the low-order 32 bits of the value.

The Fixed-Point and Fixed-Rectangle Structures

The Movie Toolbox matrix functions provide two mechanisms for specifying points and rectangles. Some of the functions work with standard QuickDraw points and rectangles, which use integer values to identify coordinates. Others, such as the `TransformFixedRect` function, work with points and rectangles whose coordinates are expressed as fixed-point numbers. By using fixed-point numbers in these points and rectangles, the Movie Toolbox can support a greater degree of precision when defining graphic objects.

The `FixedPoint` data type defines a **fixed point.** The `FixedRect` data type defines a **fixed rectangle.** Note that both of these structures define the x coordinate before the y coordinate. This is different from the standard QuickDraw structures.

```
struct FixedPoint
{
    Fixed x;        /* point's x coordinate as fixed-point number */
    Fixed y;        /* point's y coordinate as fixed-point number */
};
typedef struct FixedPoint FixedPoint;
```

Field	Description
`x`	Defines the point's x coordinate as a fixed-point number.
`y`	Defines the point's y coordinate as a fixed-point number.

```
struct FixedRect
{
    Fixed left;         /* x coordinate of upper-left corner */
    Fixed top;          /* y coordinate of upper-left corner */
    Fixed right;        /* x coordinate of lower-right corner */
    Fixed bottom;       /* y coordinate of lower-right corner */
};
typedef struct FixedRect FixedRect;
```

Field	Description
`left`	Defines the x coordinate of the upper-left corner of the rectangle as a fixed-point number.
`top`	Defines the y coordinate of the upper-left corner of the rectangle as a fixed-point number.
`right`	Defines the x coordinate of the lower-right corner of the rectangle as a fixed-point number.
`bottom`	Defines the y coordinate of the lower-right corner of the rectangle as a fixed-point number.

Media Handler Support

The video, base, and tween media handlers support sending their data to other tracks. Text data can also be sent, but none of the media handlers currently receive it. The sound, music and 3D media handlers do not support sending their data to other tracks.

Not all media handlers support all input types. Media handlers can decide which input types to support. Table 9-1 lists the input types supported by each Apple-supplied media handler.

Table 9-1 Input types supported by Apple-supplied media handlers

Input type	Video	Text	Sound	MPEG	Music	Sprite	Timecode	3D
Matrix	x	x		x		x	x	x
Graphics mode	x	x		x		x	x	x
Clip	x	x		x		x	x	x
Volume			x	x	x			
Balance			x	x	x			
Sprite image						x		x
3D sound			x		x			

Data Handler Components

QuickTime introduced a memory-based data handler. This data handler component works with movie data that is stored in memory (referenced by a handle) instead of in a file. This data handler has a component subtype value of

```
HandleDataHandlerSubType ('hndl')
```

To create a movie that uses the handle data handler, set the data reference type to `HandleDataHandlerSubType` when you call the `NewTrackMedia` function. Note that the movie data in memory is not automatically saved with the movie. If you want to save the data that is in memory, use the `FlattenMovie` or `InsertTrackSegment` functions to copy the data from memory to a file. Note that there is a special flag for `FlattenMovie` and data handlers.

The handle data handler does not use aliases as its data reference, and therefore does not use alias handles. Rather, it uses 4-byte memory handles as its data reference. The data reference contains the actual handle that stores the needed data. If you pass a handle value of `nilMovie` Data Types, the data handler allocates and manages the handle for you. If you pass a handle value other than `nilMovie` Data Types, the data handler uses your handle. It is then your responsibility to manage the handle and dispose of it when appropriate. Note that a single handle may be shared by several data handler components. Whenever new data is added, the data handler resizes the handle to accommodate new data.

Support for Progressive Downloads

The QuickTime Movie Toolbox includes support for progressive downloads, which allow part of a movie to be displayed before all of its data has been received over a network or other slow link.

Applications that use the movie controller component provided by Apple automatically get support for progressive downloads. Applications that do not use the standard movie controller can use the two high-level functions for progressive downloads, `QTMovieNeedsTimeTable` and `GetMaxLoadedTimeInMovie`, to determine whether a movie is being progressively downloaded and, if so, to see how much of it has already been downloaded. Finally, the few applications that need even more control over progressive downloads, such as control over individual tracks or media, can use one or both of the low-level functions for progressive downloads, `MakeTrackTimeTable` and `MakeMediaTimeTable`.

Displaying a Progressively Downloaded Movie

Listing 9-1 illustrates how to use the `QTMovieNeedsTimeTable` function, to find out if a movie if being progressively downloaded, and the `GetMaxLoadedTimeInMovie` function, to find out how much of the movie has been downloaded.

Listing 9-1 Displaying a progressively downloaded movie

```
WindowPtr   movieWindow;
Movie       theMovie;
Boolean     needsTimeTable;
TimeValue   loadedTime = -1;
err = GetDisplayedMovie (&movieWindow, &theMovie);
err = QTMovieNeedsTimeTable (theMovie, &needsTimeTable);
if (needsTimeTable)
{
    err = GetMaxLoadedTimeInMovie (theMovie, &loadedTime);
    // Display the movie up to the current end
}
```

Low-Level Routines

Some applications may need more control over progressive downloads, such as control over individual tracks or media, than is possible with the high-level functions for progressive downloads. These applications can use one or both of the low-level functions for progressive downloads described in this section, `MakeTrackTimeTable` and `MakeMediaTimeTable`.

Handling Media Sample References

You could always use `GetMediaSampleReference` to access samples in a movie one at a time. QuickTime introduced `GetMediaSampleReferences` (note that this is the plural form of the `GetMediaSampleReference` function), which you can use to obtain information about groups of samples. QuickTime also introduced `AddMediaSampleReferences`, which you can use to work with groups of samples that have already been added to a movie.

Manipulating Media Input Maps

The Movie Toolbox contains two functions for maintaining media input maps: `GetMediaInputMap` and `SetMediaInputMap`.

Each track has particular attributes such as size, position, and volume associated with it. The media input map of that track describes where the variable parameters are stored so that modifier tracks know where to send their data. When a track is copied, its input map is also copied. `CopyTrackSettings` also transfers the media input map.

Matrix Functions

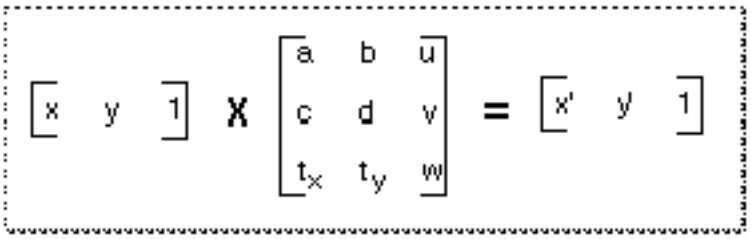
The Movie Toolbox provides a number of functions that allow you to work with transformation matrices. This chapter describes those functions.

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. 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 10-1 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 10-1 A point transformed by a 3-by-3 matrix



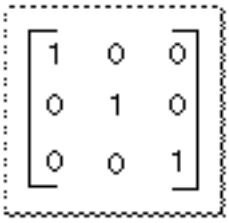
During display operations, the contents of a 3-by-3 matrix transform a point (x,y) into a point (x',y') by means of the following equations:

x' = ax + cy + t(x)

y' = bx + dy + t(y)

For example, the matrix shown in Figure 10-2 performs no transformation. It is referred to as the **identity matrix.**

Figure 10-2 The identity matrix



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):

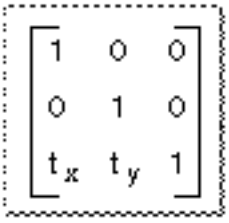
x' = 1x + 0y + 0

y' = 0x + 1y + 0

x' = y and y' = y

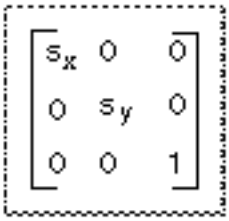
To move an image by a specified displacement, you perform a translation operation. This operation modifies the x and y coordinates of each point by a specified amount. The matrix shown in Figure 10-3 describes a translation operation.

Figure 10-3 A matrix that describes a translation operation



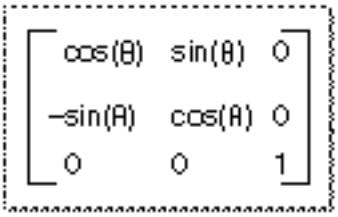
You can stretch or shrink an image by performing a scaling operation. This operation modifies the x and y coordinates by some factor. The magnitude of the x and y factors governs whether the new image is larger or smaller than the original. In addition, by making the x factor negative, you can flip the 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 10-4 describes a scaling operation.

Figure 10-4 A matrix that describes a scaling operation



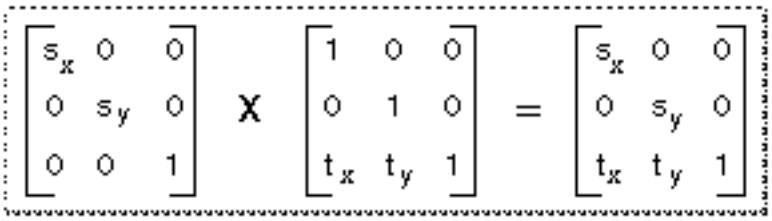
Finally, you can rotate an image by a specified angle by performing a rotation operation. You specify the magnitude and direction of the rotation by specifying factors for both x and y. The matrix shown in Figure 10-5 rotates an image counterclockwise by an angle q.

Figure 10-5 A matrix that describes a rotation operation



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 10-6.

Figure 10-6 A matrix that describes a scaling and translation operation



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.

Movie Toolbox Constants, Data Types, and Functions

This chapter describes some of the constants, data types, and functions in the Movie Toolbox that you can use in your application development. It also discusses media and data handler support in QuickTime.

Movie Exporting Flags

The `flags` parameter for the `ConvertMovieToFile` function specifies a set of movie conversion flags. QuickTime provides this flag:

```
enum {
    showUserSettingsDialog  = 2,
    movieToFileOnlyExport   = 4,
    movieFileSpecValid      = 8
};
```

Term	Definition
`showUserSettings-Dialog`	If this flag is set, the Save As dialog box will be displayed to allow the user to choose the type of file to export to, optional export settings, and the file name to export to.
`movieToFile-OnlyExport`	If this flag is set and the `showUserSettingsDialog` flag is set, the Save As dialog box restricts the user to those file formats that are supported by movie data export components. If this flag is not set, the user will also be able to save the movie either as a self-contained movie or as a reference movie.
`movieFileSpecValid`	If this flag is set and the `showUserSettingsDialog` flag is set, the `name` field of the `outputFile` parameter is used as the default name of the exported file in the Save As dialog box.

Movie Importing Flags

The `flags` parameter for the `ConvertFileToMovieFile` and `PasteHandleIntoMovie` functions specifiy a set of movie conversion flags. QuickTime provides one additional flag:

```
enum {
    showUserSettingsDialog  = 2
};
```

Term	Definition
showUserSettings-Dialog	If this flag is set, the user settings dialog box for that import operation can be displayed. For example, when importing a picture, this flag would display the standard compression dialog box so that the user could select a compression method.

Flattening Flags

The flags parameter for the FlattenMovieData function specifies a set of movie flattening flags. QuickTime provided one new flag that you must set when specifying a data reference to flatten a movie to, instead of a file:

```
enum {
    flattenFSSpecPtrIsDataRefRecordPtr  = 1L << 4
};
```

Term	Definition
flattenFSSpecPtrIs-DataRefRecordPtr	Set this flag to 1 if the FSSpec pointer is a DataReferencePtrMovie Data Types. This capability enables you to flatten movies for devices other than file systems.

Interesting Times Flags

The interestingTimeFlags parameter for the interesting time functions (GetMovieNextInterestingTime, GetTrackNexttInterestingTime, and GetMediaNextInterestingTime) specifies a set of bit flags that specify search criteria. Normally, you use one of the interesting time functions to step forward to the next frame.

These functions work well for most media types, including video and text. However, because QuickTime stores an entire MPEG stream as a single sample, stepping to the next sample skips to the end of the sequence. To solve this problem, QuickTime introduced a new flag for the interesting time calls: nextTimeStep. This flag returns the time of the next frame, even if there are multiple frames per sample, for all media types including video, text, and MPEG. Applications that implement single stepping capabilities should always use this flag instead of nextTimeMediaSample.

```
enum {
    nextTimeStep    = 1 << 4
};
```

Term	Definition
nextTimeStep	Searches for the next frame in the movie's media. Set this flag to 1 to step to the next frame.

Full Screen Flags

The `flags` parameter for the `BeginFullScreen` function specifies a set of bit flags that control certain aspects of full-screen mode. QuickTime defines these constants that you can use in the `flags` parameter:

```
enum {
    fullScreenHideCursor            = 1L << 0,
    fullScreenAllowEvents           = 1L << 1,
    fullScreenDontChangeMenuBar     = 1L << 2,
    fullScreenPreflightSize         = 1L << 3
};
```

Term	Definition
`fullScreenHideCursor`	If this flag is set, `BeginFullScreen` hides the cursor. This is useful if you are going to play a QuickTime movie and do not want the cursor to be visible over the movie.
`fullScreenAllowEvents`	If this flag is set, your application intends to allow other applications to run (by calling `WaitNextEvent` to grant them processing time). In this case, `BeginFullScreen` does not change the monitor resolution, because other applications might depend on the current resolution.
`fullScreenDont-ChangeMenuBar`	If this flag is set, `BeginFullScreen` does not hide the menu bar. This is useful if you want to change the resolution of the monitor but still need to allow the user to access the menu bar.
`fullScreenPreflight-Size`	If this flag is set, `BeginFullScreen` doesn't change any monitor settings, but returns the actual height and width that it would use if this bit were not set. This allows applications to test for the availability of a monitor setting without having to switch to it.

Text Sample Display Flags

The`displayflags` parameter for the `TextMediaAddTESample` and `TextMediaAddTextSample` functions control the behavior of the text media handler. QuickTime provides these flags:

```
enum {
    dfContinuousScroll      = 1 << 9,
    dfFlowHoriz             = 1 << 10,
    dfContinuousKaraoke     = 1 << 11,
    dfDropShadow            = 1 << 12,
    dfAntiAlias             = 1 << 13,
    dfKeyedText             = 1 << 14,
    dfInverseHilite         = 1 << 15,
    dfTextColorHilite       = 1 << 16
};
```

Term	Definition
dfContinuousScroll	If this flag is set, the text media handler lets new samples cause previous samples to scroll out. You must also set dfScrollIn or dfScrollOffMovie Data Types, or both, for this to take effect.
dfFlowHoriz	If this flag is set, the text media handler lets horizontally scrolled text flow within the text box instead of extending to the right.
dfContinuousKaraoke	If this flag is set, the text media handler ignores the starting offset when highlighting text. Instead, it highlights text from the beginning of the text sample to the ending offset.
dfDropShadow	If this flag is set, the text media handler displays text with a drop shadow. If you use the TextMediaSetTextSampleData function, the position and translucency of the drop shadow is under your application's control.
dfAntiAlias	If this flag is set, the text media handler displays text with anti-aliasing. Note that although anti-aliased text looks smoother, anti-aliasing can slow down performance.
dfKeyedText	If this flag is set, the text media handler renders text over the background without drawing the background color. This technique is also known as masked text.
dfInverseHilite	If this flag is set, the text media handler highlights text using inverse video instead of the highlight color.
dfTextColorHilite	If this flag is set, the text media handler highlights text by changing the color of the text.

Data Types

Most Movie Toolbox data structures are private data structures. Your application never modifies the contents of these structures directly. Rather, the Movie Toolbox provides a number of functions that allow you to work with these data structures.

Modifier Input Types

The media input map describes the meaning of each input to a track. Each track has particular attributes associated with it, such as size, position, and volume. The media input map of that track describes the mapping of track modifier inputs to track properties. When you want to modify the attributes of a track, you can insert a track modifier input such as kTrackModifierTypeMatrix into the input map. The values stored in the modifier input you inserted will affect the values that are currently stored with the track.

Custom media handlers can define additional input types as necessary. Apple Computer reserves all input types consisting entirely of lowercase letters.

The following input types are currently defined:

```
enum {
    kTrackModifierTypeMatrix            = 1,
    kTrackModifierTypeClip              = 2,
    kTrackModifierTypeGraphicsMode      = 5,
    kTrackModifierTypeVolume            = 3,
    kTrackModifierTypeBalance           = 4,
    kTrackModifierTypeImage             = 'vide',
    kTrackModifierObjectMatrix          = 6,
    kTrackModifierObjectGraphicsMode    = 7,
    kTrackModifierType3d4x4Matrix       = 8,
    kTrackModifierCameraData            = 9
    kTrackModifierSoundLocalization     = 10
};
```

Term	Definition
`kTrackModifier-TypeMatrix`	Data sent to this input should be in the form of a QuickTime `MatrixRecordMovie` Data Types. The matrix is concatenated with the track and movie matrices to determine the tracks final location and size. The matrix modifier describes relative, not absolute, position and scaling.
`kTrackModifier-TypeClip`	Data sent to this input should be in the form of a QuickDraw region. The region is intersected with the track's source box.
`kTrackModifierType-GraphicsMode`	Data sent to this input should be in the form of a `ModifierTrack-GraphicsModeRecord` data type. The contents of the record are used as the graphics mode setting for the track. The graphics mode is not combined with the track's current graphics mode, but rather overrides it.
`kTrackModifier-TypeVolume`	Data sent to this input should be in the form of a 16-bit fixed-point number. This is the same format in which QuickTime sound volume levels are stored. The volume level is used as a scaling factor on the sound track's level. It is multiplied with the track and movie volumes to determine the track's overall volume.
`kTrackModifier-TypeBalance`	Data sent to this input should be in the form of a 8-bit fixed-point number. This is the same format in which QuickTime balance values are stored. The balance value is used as the balance setting for the track. Unlike the volume modifier, it is not concatenated with the track's current balance level, but overrides the current balance level.
`kTrackModifier-TypeImage`	Data sent to this input should be compressed video data, typically from a video track. This input type can be used with sprite tracks. For sprite tracks, the image data is used to replace the image of a specified image index in the sprite track. The index of the image to replace must be specified in the media input map when the reference is created.
`kTrackModifier-ObjectMatrix`	Data sent to this input should be in the form of a QuickTime `MatrixRecordMovie` Data Types. The matrix is sent to a particular object within the receiving track, as specified by the `kTrackModifierObjectID` atom in the input map. The matrix acts as an override to the object's current matrix. For example, the matrix could be sent to a sprite within a sprite track. It would cause the sprite to move, not the entire sprite track as would `kTrackModifierMatrixMovie` Data Types.

Term	Definition
kTrackModifierObject-GraphicsMode	Data sent to this input should be in the form of a ModifierTrack-GraphicsModeRecord data type. The contents of the record are used to vary the opacity of an object within the track. For example, you would use data sent to this input to vary the opacity of a sprite within a sprite track, rather than modifying the opacity of the entire sprite track.
kTrackModifierSound-Localization	Data sent to this input should be in the form of a sound localization data record (SSpLocalizationDataMovie Data Types). This data overrides the sound localization settings already in use by the track.

Data References

The Movie Toolbox fully supports a media that refers to data in more than one file. By allowing a single media to refer to more than one file, the toolbox allows better playback performance and easier editing, primarily by reducing the number of tracks in a movie. Use the SetMediaDefaultDataRefIndex function to control which of a media's files you access when you add new sample data.

To fully specify a data reference, it is necessary to provide the data reference itself, along with its type; the data reference handle does not contain the type of the data reference. The DataReferenceRecord data structure contains both of these pieces of information, making it possible to pass them to functions as a single parameter. The FlattenMovieData function uses the information in the data reference structure to flatten a movie to a data reference instead of to a file.

```
struct DataReferenceRecord {
    OSType  dataRefType;
    Handle  dataRef;
};
typedef struct DataReferenceRecord DataReferenceRecord;
typedef DataReferenceRecord *DataReferencePtr;
```

Field	Definition
dataRefType	Specifies the type of data reference. For an alias data reference, you set the parameter to rAliasType, indicating that the reference is an alias. For a handle data reference, set the parameter to HandleDataHandlerSubTypeMovie Data Types.
dataRef	Specifies the actual data reference. This parameter contains a handle to the information that identifies the file to be used. The type of information stored in the handle depends on the value of the dataRefType parameter. For example, if your application is loading the movie from a file, this parameter would contain an alias to the movie file.

Movie Identifiers

You identify a data structure to the Movie Toolbox by means of a data type that is supplied by the Movie Toolbox. The following data types are currently defined:

Type	Definition
`Media`	Specifies the media for an operation. Your application obtains a media identifier from such Movie Toolbox functions as `NewTrackMedia` and `GetTrackMedia`.
`Movie`	Specifies the movie for an operation. Your application obtains a movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle`.
`MovieEditState`	Specifies the movie edit state for an operation. Your application obtains a movie edit state identifier when you create the edit state by calling the `NewMovieEditState` function.
`QTCallBack`	Specifies the callback for an operation. You obtain a callback identifier from the `NewCallBack` function.
`TimeBase`	Specifies the time base for an operation. Your application obtains a time base identifier from the `NewTimeBase` or `GetMovieTimeBase` functions.
`Track`	Specifies the track for an operation. Your application obtains a track identifier from such Movie Toolbox functions as `NewMovieTrack` and `GetMovieTrack`.
`TrackEditState`	Specifies the track edit state for an operation. Your application obtains a track edit state identifier when you create the edit state by calling the `NewTrackEditState` function.
`UserData`	Specifies the user data list for an operation. You obtain a user data list identifier by calling the `GetMovieUserData`, `GetTrackUserData`, or `GetMediaUserData` functions.

Constants

This section lists the constants that were added to QuickTime after an early release. Most of the constants are used as flags for QuickTime functions; they allow the programmer to pass numeric data as a meaningful name.

```
#define kFix1 = (0x00010000);              /* fixed point value equal to 1.0 */
#define gestaltQuickTime 'qtim'            /* Movie Toolbox availability */
#define MovieFileType 'MooV'               /* movie file type */
#define VideoMediaType 'vide'              /* video media type */
#define SoundMediaType 'soun'              /* sound media type */
#define MediaHandlerType 'mhlr'            /* media handler type */
#define DataHandlerType 'dhlr'             /* data handler type */
#define TextMediaType 'text'               /* text media type */
#define GenericMediaType 'gnrc'            /* base media handler type */
#define DoTheRightThing = 0L               /* indicates default flag settings
                                                 for Movie Toolbox functions */

/* sound volume values in trackVolume parameter of NewMovieTrack function */
#define kFullVolume         = 0x100        /* full, natural volume
                                              (8.8 format) */
#define kNoVolume           =   0          /* no volume */
/* constants for whichMediaTypes parameter of GetMovieNextInterestingTime
```

```
    function */
#define VisualMediaCharacteristic 'eyes'  /* visual media */
#define AudioMediaCharacteristic 'ears'   /* audio media */

enum
{
/* media quality settings in quality parameter of SetMediaQuality function */
    mediaQualityDraft           = 0x0000,    /* lowest quality level */
    mediaQualityNormal          = 0x0040,    /* acceptable quality level */
    mediaQualityBetter          = 0x0080,    /* better quality level */
    mediaQualityBest            = 0x00C0     /* best quality level */
};

/*values for callBackFlags field of QuickTime callback header structure used
    by clock components to communicate scheduling information about a
    callback event to the Movie Toolbox */
enum
{
    qtcbNeedsRateChanges           = 1,    /* rate changes */
    qtcbNeedsTimeChanges           = 2,    /* time changes */
    qtcbNeedsStartStopChanges      = 4     /* time base changes at start &
                                              stop times */
};

/* dialog items to include in dialog box definition for use with
    SFPGetFilePreview function */
enum
{
    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 */
};

enum
{
    movieInDataForkResID           = -1        /* magic resource ID */
};

/* flags for LoadIntoRAM functions */
enum
{
    keepInRam                   = 1<<0,     /* load and make so data
                                               cannot be purged */
    unkeepInRam                 = 1<<1,     /* mark data so it can be purged */

    flushFromRam                = 1<<2,     /* empty handles and purge data
                                               from memory */
    loadForwardTrackEdits      = 1<<3,     /* load only data around track edits;
                                               play movie forward */
    loadBackwardTrackEdits      = 1<<4      /* load only data around edits;
                                               play movie in reverse */
};

/* flag for PasteHandleIntoMovie function */
enum
{
```

```
    pasteInParallel = 1         /* changes function to take contents and type
of
                                    handle and add to movie */
};

/* text description display flags used in TextMediaAddTextSample and
      TextMediaAddTESample */
enum
{
    dfDontDisplay           = 1<<0,   /* don't display the text */
    dfDontAutoScale         = 1<<1,   /* don't scale text as track
                                         bounds grows or shrinks */
    dfClipToTextBox         = 1<<2,   /* clip update to the text box */
    dfUseMovieBGColor       = 1<<3,   /* set text background to movie's
                                         background color */
    dfShrinkTextBoxToFit    = 1<<4,   /* compute minimum box to fit the
                                         sample */
    dfScrollIn              = 1<<5,   /* scroll text in until last of
                                         text is in view */
    dfScrollOut             = 1<<6    /* scroll text out until last of text is
                                         gone (if dfScrollIn is also set,
                                         scroll in then out) */
    dfHorizScroll           = 1<<7,   /* scroll text horizontally; otherwise,
                                         it's vertical */
    dfReverseScroll         = 1<<8    /* vertically scroll down and horizontally
                                        scroll up; justification-dependent */

};

/* find flags for TextMediaFindNextText function */
    findTextEdgeOK          = 1<<0,     /* OK to find text at specified
                                           sample time */
    findTextCaseSensitive   = 1<<1,     /* case-sensitive search */
    findTextReverseSearch   = 1<<2,     /* search from sampleTime backward */
    findTextWraparound      = 1<<3,     /* wrap search when beginning or end
                                           of movie is reached */

/* return display flags for application-defined text function */
enum
{
    txtProcDefaultDisplay   = 0,        /* use the media's default settings */
    txtProcDontDisplay      = 1,        /* don't display the text */
    txtProcDoDisplay        = 2         /* display the text */
};

enum
{
hintsScrubMode              = 1<<0,    /* toolbox can display key frames when
                                          movie is repositioned */
hintsAllowInterlace         = 1<<6,    /* use interlace option for compressor
                                          components */
hintsUseSoundInterp         = 1<<7     /* turn on sound interpolation */
};
typedef unsigned long playHintsEnum;
```

Result Codes

The following table shows common error codes returned by Movie Toolbox functions.

Constant	Value	Description
couldNotResolveDataRef	-2000	Cannot use this data reference
badImageDescription	-2001	Problem with this image description
badPublicMovieAtom	-2002	Movie file corrupted
cantFindHandler	-2003	Cannot locate this handler
cantOpenHandler	-2004	Cannot open this handler
badComponentType	-2005	Component cannot accommodate this data
noMediaHandler	-2006	Media has no media handler
noDataHandler	-2007	Media has no data handler
invalidMedia	-2008	This media is corrupted or invalid
invalidTrack	-2009	This track is corrupted or invalid
invalidMovie	-2010	This movie is corrupted or invalid
invalidSampleTable	-2011	This sample table is corrupted or invalid
invalidDataRef	-2012	This data reference is invalid
invalidHandler	-2013	This handler is invalid
invalidDuration	-2014	This duration value is invalid
invalidTime	-2015	This time value is invalid
cantPutPublicMovieAtom	-2016	Cannot write to this movie file
badEditList	-2017	The track's edit list is corrupted
mediaTypesDontMatch	-2018	These media don't match
progressProcAborted	-2019	Your progress procedure returned an error
movieToolboxUninitialized	-2020	You haven't initialized the Movie Toolbox
wfFileNotFound	-2021	Cannot locate this file
cantCreateSingleForkFile	-2022	Error trying to create a single-fork file. This occurs when the file already exists.
invalidEditState	-2023	This edit state is invalid

Constant	Value	Description
nonMatchingEditState	-2024	This edit state is not valid for this movie
staleEditState	-2025	Movie or track has been disposed
userDataItemNotFound	-2026	Cannot locate this user data item
maxSizeToGrowTooSmall	-2027	Maximum size must be larger
badTrackIndex	-2028	This track index value is not valid
trackIDNotFound	-2029	Cannot locate a track with this ID value
trackNotInMovie	-2030	This track is not in this movie
timeNotInTrack	-2031	This time value is outside of this track
timeNotInMedia	-2032	This time value is outside of this media
badEditIndex	-2033	This edit index value is not valid
internalQuickTimeError	-2034	Internal error
cantEnableTrack	-2035	Cannot enable this track
invalidRect	-2036	Specified rectangle has invalid coordinates
invalidSampleNum	-2037	There is no sample with this sample number
invalidChunkNum	-2038	There is no chunk with this chunk number
invalidSampleDescIndex	-2039	Sample description index value invalid
invalidChunkCache	-2040	The chunk cache is corrupted
invalidSampleDescription	-2041	This sample description is invalid or corrupted
dataNotOpenForRead	-2042	Cannot read from this data source
dataNotOpenForWrite	-2043	Cannot write to this data source
dataAlreadyOpenForWrite	-2044	Data source is already open for write
dataAlreadyClosed	-2045	You have already closed this data source
endOfDataReached	-2046	End of data
dataNoDataRef	-2047	No data reference value found
noMovieFound	-2048	Toolbox cannot find a movie in the movie file
invalidDataRefContainer	-2049	Invalid data reference
badDataRefIndex	-2050	Data reference index value is invalid

Constant	Value	Description
noDefaultDataRef	-2051	Could not find a default data reference
couldNotUseAnExistingSample	-2052	Movie Toolbox could not use a sample
featureUnsupported	-2053	Movie Toolbox does not support this feature

Document Revision History

This table describes the changes to *QuickTime Movie Basics*.

Date	Notes
2006-01-10	New document that introduces basic concepts underlying QuickTime movies.
	Replaces "Movie Toolbox: Editing," "Movie Toolbox: Application-Defined Functions," "Movie Toolbox: Data Types," and "Movie Toolbox: Saving Movies."
2002-09-17	New document that explains how to enable the editing of movies.

Document Revision History