INSIDE MACINTOSH

QuickTime

Addison-Wesley Publishing Company

Reading, Massachusetts Menlo Park, California New York
Don Mills, Ontario Wokingham, England Amsterdam Bonn
Sydney Singapore Tokyo Madrid San Juan
Paris Seoul Milan Mexico City Taipei

Apple Computer, Inc.
20525 Mariani Avenue
Cupertino, CA 95014
408-996-1010

Simultaneously published in the United States and Canada.

LIMITED WARRANTY ON MEDIA AND REPLACEMENT

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

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

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS MANUAL, even if advised of the possibility of such damages.

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

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

ISBN 0-201-62201-7
1 2 3 4 5 6 7 8 9-MU-9796959493
First Printing, March 1993

Contents

Figures, Tables, and Listings

Chapter 1 Introduction to QuickTime 1-1

Chapter 2 Movie Toolbox 2-1

About This Book

This book describes QuickTime, an extension of Macintosh system software that enables you to integrate time-based data into mainstream Macintosh applications. This book also provides a complete technical reference to the Movie Toolbox, the Image Compression Manager, and the movie resources.

Time-based data types contain data that can be stored and retrieved as values over time. Examples include sound, video, animation, data produced by scientific instruments, and financial results. Time-based data can now be manipulated in the same ways as other standard types of data in the Macintosh environment. In QuickTime, a set of time-based data is referred to as a **movie.** This book shows in detail how your application can allow users to display, edit, cut, copy, and paste movies and movie data in the same way that they can work with text and graphic elements today.

If you want your application to be able to handle time-based data, you should first read the chapter "Introduction to QuickTime" for an introduction to the QuickTime concepts, architecture, managers, and components.

If you want your application to be able to paste and run QuickTime movies, to edit them, or to create new movies, you should read the chapter "Movie Toolbox." Your application may only need to paste a movie from the Clipboard and play it—for example, a word processor might paste a movie as it does a picture, and the user might use a movie controller to play the movie. A more media-intensive application might add the ability to edit the movie after it is pasted—for example, the user might cut a segment of the movie, add a video segment, or add a different sound track. Full "mediagenic" applications could create a movie from disparate sources such as CD tracks, video clips, sounds, animation from graphics programs, or still images.

If you want your application to use the facilities of QuickTime to compress and decompress still images, you should read the chapter "Image Compression Manager." These single images are not QuickTime movies—they do not contain time-based data. Nevertheless, you can use the image compression and decompression facilities of QuickTime for images that are not stored in movies. The chapter describes the Image Compression Manager, including compression and decompression algorithms, and the steps involved in compressing and decompressing single images and sequences of images.

If you are going to play movies or compress images, you should be familiar with QuickDraw and Color QuickDraw, described in *Inside Macintosh: Imaging*. If you are going to create QuickTime movies, you should also be familiar with the Sound Manager, described in *Inside Macintosh: More Macintosh Toolbox*, and with the human interface guidelines as described in *Macintosh Human Interface Guidelines*. If you are going to use QuickTime

components, you should be familiar with the Component Manager as described in *Inside Macintosh: More Macintosh Toolbox*.

If your application imports or exports movies to other platforms, you should read the chapter "Movie Resource Formats." It presents details of the movie file format used by QuickTime. Most applications do not need this information.

The companion to this book, *Inside Macintosh: QuickTime Components*, includes descriptions of the Apple-supplied QuickTime components: clock components, compressor components, standard image-compression dialog components, movie controller components, sequence grabber components, sequence grabber channel components, sequence grabber panel components, video digitizer components, media data-exchange components, preview components, and media handler components.

Format of a Typical Chapter

Almost all chapters in this book follow a standard structure. For example, the chapter "Image Compression Manager" contains these sections:

- "Introduction to the Image Compression Manager." This section presents a general introduction to image compression.
- "About Image Compression." This section provides an overview of the features provided by the Image Compression Manager.
- "Using the Image Compression Manager." This section describes the tasks you can accomplish using the Image Compression Manager. It describes how to use the most common functions, gives related user interface information, provides code samples, and supplies additional information.
- "Image Compression Manager Reference." This section provides a complete reference to the Image Compression Manager by describing the constants, data structures, and functions that it uses. Each function description also follows a standard format, which gives the function declaration and description of every parameter of the function. Some function descriptions also give additional descriptive information, such as assembly-language information or result codes.
- "Summary of the Image Compression Manager." This section provides the Image Compression Manager's C interface, as well as the Pascal interface, for the constants, data structures, functions, and result codes associated with the Image Compression Manager.

Conventions Used in This Book

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

Special Fonts

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

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

Types of Notes

There are several types of notes used in this book.

Note

A note like this contains information that is interesting but possibly not essential to an understanding of the main text. (An example appears on page 1-3.) u

IMPORTANT

A note like this contains information that is essential for an understanding of the main text. (An example appears on page 2-84.) s

s **WARNING**

Warnings like this indicate potential problems that you should be aware of as you design your application. Failure to heed these warnings could result in system crashes or loss of data. (An example appears on page 2-59.) s

Development Environment

The system software functions described in this book are available using C or Pascal interfaces. How you access these functions depends on the development environment you are using. This book shows system software functions in their C interface using the Macintosh Programmer's Workshop (MPW) version 3.2.

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

In a few cases, the functions documented in one chapter may be listed in the MPW interface files associated with another manager. An example is the `MakeFilePreview` function, which is documented for conceptual consistency in the chapter "Movie Toolbox." This function does not appear in the `Movies.h` MPW interface file; rather, it is listed in the `ImageCompression.h` MPW interface file. When this occurs, the disparity is noted in the function descriptions.

APDA, Apple's source for developer tools, offers worldwide access to a broad range of programming products, resources, and information for anyone developing on Apple platforms. You'll find the most current versions of Apple and third-party development tools, debuggers, compilers, languages, and technical references for all Apple platforms. To establish an APDA account, obtain additional ordering information, or find out about site licensing and developer training programs, contact

APDA
Apple Computer, Inc.
P. O. Box 319
Buffalo, NY 14207-0319

Telephone:	800-282-2732 (United States)
	800-637-0029 (Canada)
	716-871-6555 (International)
Fax:	716-871-6511
AppleLink:	APDA
America Online:	APDA
CompuServe:	76666,2405
Internet:	APDA@applelink.apple.com

If you provide commercial products and services, call 408-974-4897 for information on the developer support programs available from Apple.

For information on registering signatures, file types, Apple events, and other technical information, contact

Macintosh Developer Technical Support
Apple Computer, Inc.
20525 Mariani Avenue, M/S 75-3T
Cupertino, CA 95014-6299

CHAPTER 1

Introduction to QuickTime

Contents

This chapter introduces the concepts underlying QuickTime, a set of functions and data structures that you can use in your application to control time-based data. In QuickTime, a set of time-based data is referred to as a *movie*. Your application can allow users to display, edit, cut, copy, and paste movies and movie data in the same way that they can work with text and graphic elements today.

This chapter also introduces the QuickTime architecture, the managers, and the components that constitute QuickTime. It will help you decide what level of QuickTime support your application may need to incorporate.

QuickTime Concepts

To use QuickTime, you need to understand some concepts that are new to most developers of Macintosh applications: movies, media data structures, components, image compression, and time.

Movies and Media Data Structures

A traditional movie, whether stored on film, laser disk, or tape, is a continuous stream of data. A QuickTime movie can be similarly constructed, but it need not be: a QuickTime movie can consist of data in sequences from different forms, such as analog video and CD-ROM. The movie is not the medium; it is the organizing principle.

A QuickTime movie may contain several **tracks.** Each track refers to a media that contains references to the movie data, which may be stored as images or sound on hard disks, floppy disks, compact discs, or other devices. The data references constitute the track's **media.** Each track has a single media data structure.

Note

Throughout this book, the term *media* is used to refer to a Movie Toolbox data structure that contains information that describes the data for a track in a movie. Note that a media does not contain its data; rather, a media contains a reference to its data. If more than one media is being discussed, the term *media structures* is used. u

Your application need never work directly with the movie data, as Movie Toolbox functions allow you to manage movie content and characteristics. See the chapter "Movie Toolbox" later in this book for a comprehensive reference to the Movie Toolbox.

Components

QuickTime provides components so that every application doesn't need to know about all possible types of audio, visual, and storage devices. A **component** is a code resource that is registered by the Component Manager. The component's code can be available as a systemwide resource or in a resource that is local to a particular application. Each QuickTime component supports a defined set of features and presents a specified

functional interface to its client applications. Applications are thereby isolated from the details of implementing and managing a given technology. For example, you could create a component that supports a certain data encryption algorithm. Applications could then use your algorithm by connecting to your component through the Component Manager, rather than by implementing the algorithm over again. For comprehensive reference to the QuickTime components supplied by Apple, see the book *Inside Macintosh: QuickTime Components*.

Image Compression

Image data requires a large amount of storage space. Storing a single 640-by-480 pixel image in 32-bit color can require as much as 1.2 MB. Similarly, sequences of images, like those that might be contained in a QuickTime movie, demand substantially more storage than single images. This is true even for sequences that consist of fairly small images, because the movie consists of a large number of those images. Consequently, minimizing the storage requirements for image data is an important consideration for any application that works with images or sequences of images.

The Image Compression Manager provides your application with an interface for compressing and decompressing images and sequences of images that is independent of devices and algorithms. See the chapter “Image Compression Manager” later in this book for details.

Time

Image compression is difficult but worthwhile—images, not to mention long sequences of images, take a lot of memory. Time management in QuickTime is equally essential. You must understand time management to understand the QuickTime functions and data structures.

Seemingly simple issues prove interesting—for example, determining the proper length (duration) of a movie. For many movies, the proper duration is the time required to play them in “real” time—that is, a rate in which human actions appear natural, and objects fall to earth accelerating at 32 feet per second per second. But what is the length of a movie that shows spreadsheet data charted over time, or a map of the earth that recapitulates continental drift? Add to this the differing clock speeds of different platforms, and the need to decompress in real time, and time proves, as ever, complex.

To manage these situations, QuickTime defines **time coordinate systems,** which anchor movies and their media data structures to a common temporal reality, the second. A time coordinate system contains a **time scale** that provides the translation between real time and the time in a movie. Time scales are marked in **time units.** The number of units that pass per second quantifies the scale—that is, a time scale of 26 means that 26 units pass per second and each time unit is 1/26 of a second. A time coordinate system also contains a **duration,** which is the length of a movie or a media in the number of time units it contains. Particular points in a movie can be identified by a time value, the number of time units elapsed to that point.

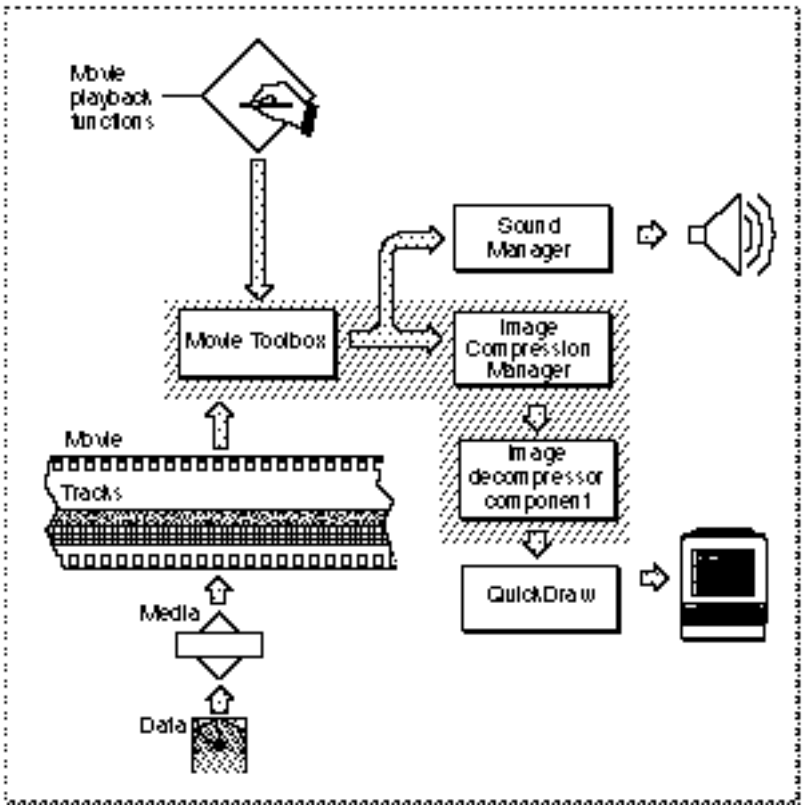
Each media has its own time coordinate system, which starts at time 0. The Movie Toolbox maps each type of media data from the movie's time coordinate system to the media's time coordinate system.

Time bases and time coordinate systems are described in the chapter "Movie Toolbox" later in this book.

The QuickTime Architecture

QuickTime comprises two managers: the Movie Toolbox and the Image Compression Manager. QuickTime also relies on the Component Manager, as well as a set of predefined components. Figure 1-1 shows the relationships of these managers and an application that is playing a movie.

Figure 1-1 QuickTime playing a movie



The following sections discuss these managers in more detail.

The Movie Toolbox

Your application gains access to the capabilities of QuickTime by calling functions in the Movie Toolbox. The Movie Toolbox allows you to store, retrieve, and manipulate time-based data that is stored in QuickTime movies. A single movie may contain several types of data. For example, a movie that contains video information might include both video data and the sound data that accompanies the video.

The Movie Toolbox also provides functions for editing movies. For example, there are editing functions for shortening a movie by removing portions of the video and sound tracks, and there are functions for extending it with the addition of new data from other QuickTime movies.

The Movie Toolbox is described in the chapter "Movie Toolbox" later in this book. That chapter includes code samples that show how to play movies.

The Image Compression Manager

The Image Compression Manager comprises a set of functions that compress and decompress images or sequences of graphic images.

The Image Compression Manager provides a device-independent and driver-independent means of compressing and decompressing images and sequences of images. It also contains a simple interface for implementing software and hardware image-compression algorithms. It provides system integration functions for storing compressed images as part of PICT files, and it offers the ability to automatically decompress compressed PICT files on any QuickTime-capable Macintosh computer.

In most cases, applications use the Image Compression Manager indirectly, by calling Movie Toolbox functions or by displaying a compressed picture. However, if your application compresses images or makes movies with compressed images, you will call Image Compression Manager functions.

The Image Compression Manager is described in the chapter "Image Compression Manager" later in this book. This chapter also includes code samples that show how to compress images or make movies with compressed images.

The Component Manager

Applications gain access to components by calling the Component Manager. The Component Manager allows you to define and register types of components and communicate with components using a standard interface. A component is a code resource that is registered by the Component Manager. The component's code can be stored in a systemwide resource or in a resource that is local to a particular application.

Once an application has connected to a component, it calls that component directly. If you create your own component class, you define the function-level interface for the component type that you have defined, and all components of that type must support the interface and adhere to those definitions. In this manner, an application can freely choose among components of a given type with absolute confidence that each will work.

The Component Manager is described in *Inside Macintosh: More Macintosh Toolbox*.

QuickTime Components

QuickTime includes several components that are provided by Apple. These components provide essential services to your application and to the managers that make up the QuickTime architecture. The following Apple-defined components are among those used by QuickTime:

- movie controller components, which allow applications to play movies using a standard user interface
- standard image-compression dialog components, which allow the user to specify the parameters for a compression operation by supplying a dialog box or a similar mechanism
- image compressor components, which compress and decompress image data
- sequence grabber components, which allow applications to preview and record video and sound data as QuickTime movies
- video digitizer components, which allow applications to control video digitization by an external device
- media data-exchange components, which allow applications to move various types of data in and out of a QuickTime movie
- derived media handler components, which allow QuickTime to support new types of data in QuickTime movies
- clock components, which provide timing services defined for QuickTime applications
- preview components, which are used by the Movie Toolbox's standard file preview functions to display and create visual previews for files
- sequence grabber components, which allow applications to obtain digitized data from sources that are external to a Macintosh computer
- sequence grabber channel components, which manipulate captured data for a sequence grabber component
- sequence grabber panel components, which allow sequence grabber components to obtain configuration information from the user for a particular sequence grabber channel component

These components and the interfaces they support are discussed in *Inside Macintosh: QuickTime Components*.

Using QuickTime

Applications that use QuickTime fall into two categories: applications that can play existing movies, and applications that can create and edit movies. The following sections describe how applications of both types use QuickTime.

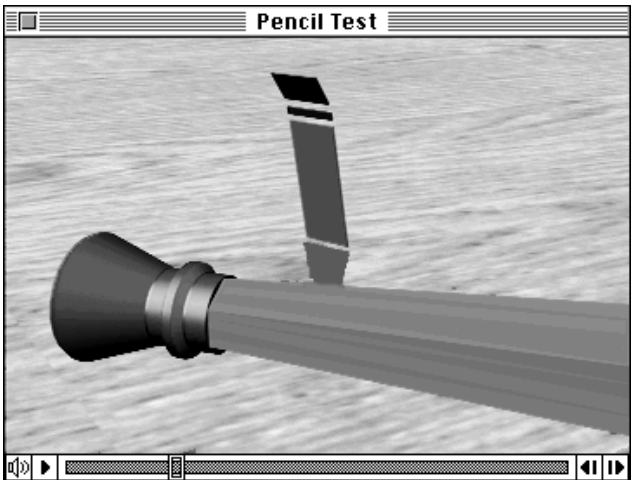
Playing Movies

QuickTime provides a complete set of tools that allow you to play movies in your application. You can also allow the user to position, resize, copy, and paste movies within the documents that your application creates and manipulates.

The Movie Toolbox provides functions that enable you to get a movie into your application; you can either get a movie from a file or from the scrap. Positioning the movie within a document varies with the application. For example, in a text document a movie might be repositioned with tab settings, whereas in a paint document the user might position the movie by selecting and dragging the movie rectangle.

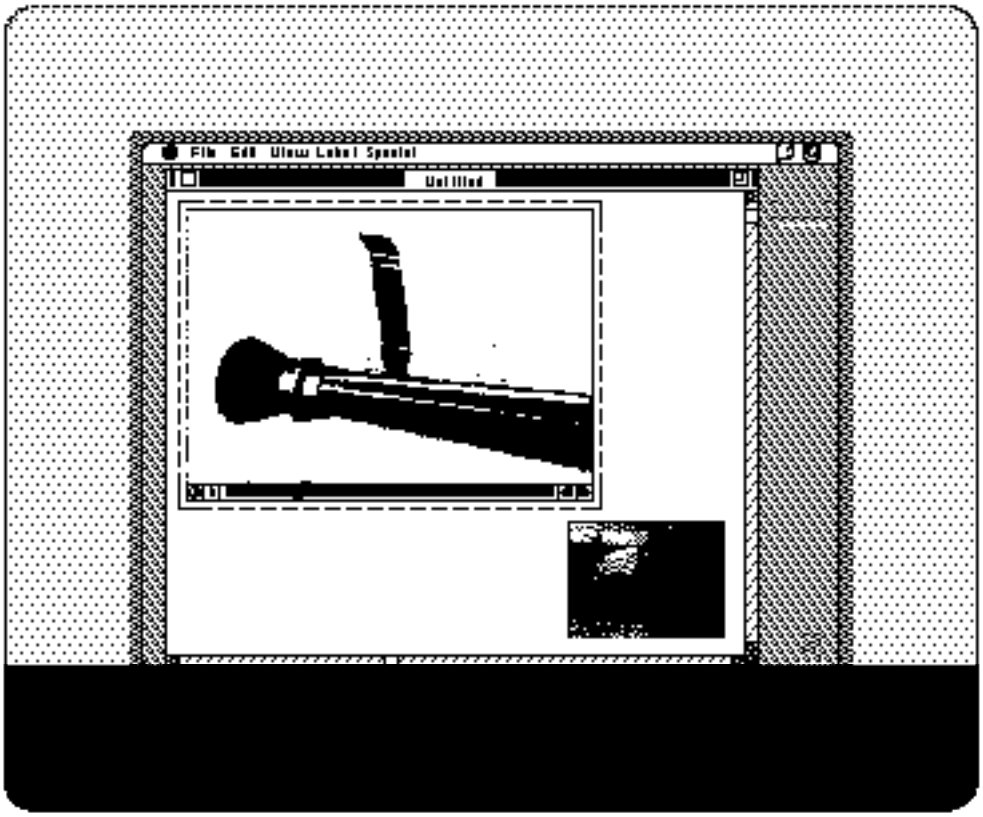
Once you have loaded the movie into your document, you can allow the user to play it by calling the movie controller component provided by Apple. Figure 1-2 shows a sample movie controller.

Figure 1-2 A QuickTime movie with Apple's movie controller



Resizing the movie's rectangle is the same as resizing PICT rectangles within a text or paint document. When the user selects the movie, a selection rectangle appears with resizing handles at the corners of the rectangle, like those shown in Figure 1-3. The user can drag the handles to resize the movie rectangle.

Figure 1-3 A QuickTime movie with an active selection rectangle



Changing the size of a movie window may affect the performance of the video during playback as well as its appearance on the display.

Creating and Editing Movies

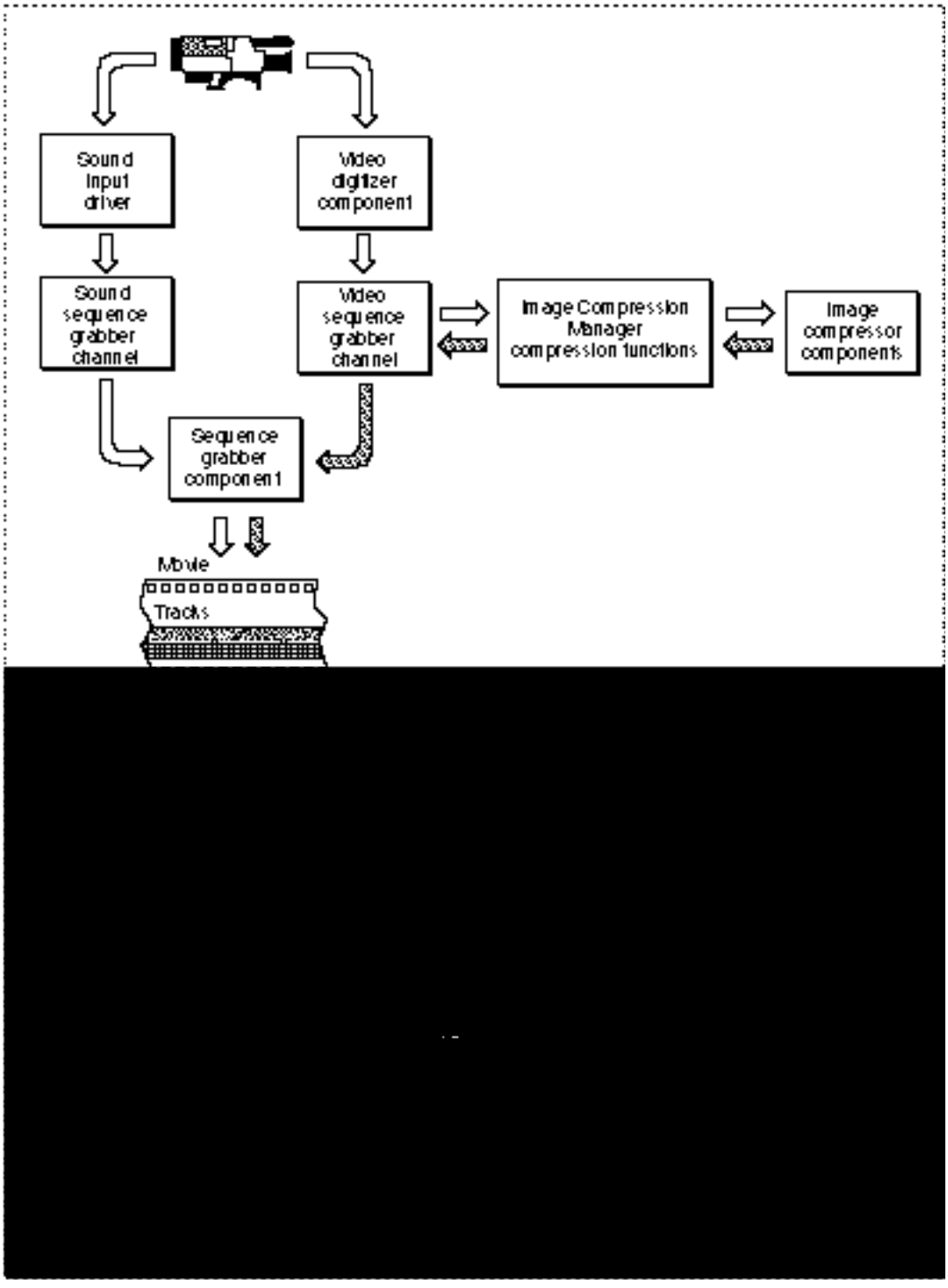
More sophisticated applications allow the user to create new movies and edit existing ones. An example of a movie-creating application is an electronic mail system that supports the creation and transmission of video memos. Other examples are an application that might be included in a video digitizer card package, an architectural walk-through program, or an application that creates animation sequences that can be saved as QuickTime movies.

Movie-creating applications fall into two categories:

- those that use a sequence grabber component and the compression functions of the Image Compression Manager to obtain movie data
- those that make a movie and then use the Movie Toolbox and the decompression functions of the Image Compression Manager to work with the movie data

If you are creating an application that creates or edits movies, you are going to use more of the capabilities of the Movie Toolbox and the other managers that make up QuickTime. Figure 1-4 shows some of these other elements in an expanded view of the QuickTime architecture. For comprehensive information on the video digitizer component, the sequence grabber channel component, the sequence grabber component, and video and media handlers, see *Inside Macintosh: QuickTime Components*.

Figure 1-4 Capturing and playing back movies



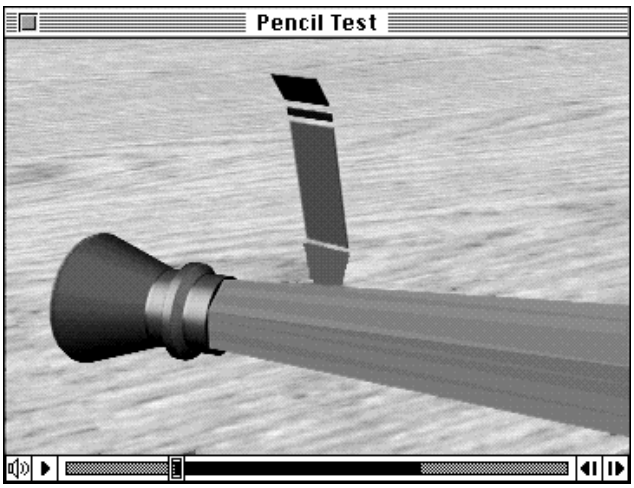
Movie-Editing Applications

The Movie Toolbox includes functions that help your application provide movie-editing capabilities to the user. The easiest way to allow the user to edit a movie is to use the movie controller component provided by Apple.

Alternatively, you can use QuickTime's editing functions to remove, copy, replace, rearrange, or extend the content of movies. The user interface for editing is up to you, as long as you observe the guidelines suggested by Apple (see the chapter "Movie Toolbox" later in this book for more information on human interface guidelines for movie applications).

To give a user some simple editing tools, you could use the movie controller component to create a movie-editing window similar to the one shown in Figure 1-5.

Figure 1-5 Apple's movie controller with a portion of the movie selected for editing



This window gives the user access to various viewing and editing controls. These controls include a real-time position controller that allows random access over the length of the movie, single-step controls in both forward and reverse directions, visual feedback for selecting a sequence of frames in the movie, and a rectangular marker highlighting the currently displayed frame.

Movie-Creating Applications

Applications that create QuickTime movies can capture the movie's data from an external source and store it in a media. As with any movie, this data may be digitized video, digitized sound, computer animation, MIDI (Musical Instrument Digital Interface) data, external data such as an audio CD or videotape, and so on. Each type of data in a movie has an associated movie track. Movie tracks contain an edit list that sequences the data stored in the media.

The Movie Toolbox supplies functions that allow you to modify the edit list of the tracks in a movie to rearrange, remove, and extend the playback display sequence of the data in the movie. You can use these functions to create an application that captures external video and creates movies.

Figure 1-6 shows a sample user interface for a video-capture application. Before the user digitizes the data, the application displays an editing window (called a *monitor window*) to help preview the information prior to capturing it.

Figure 1-6 A monitor window

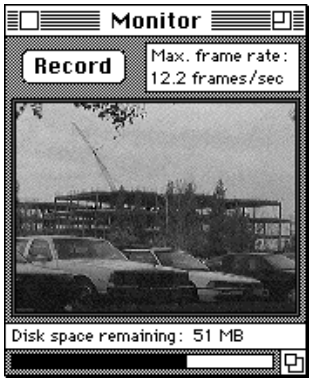
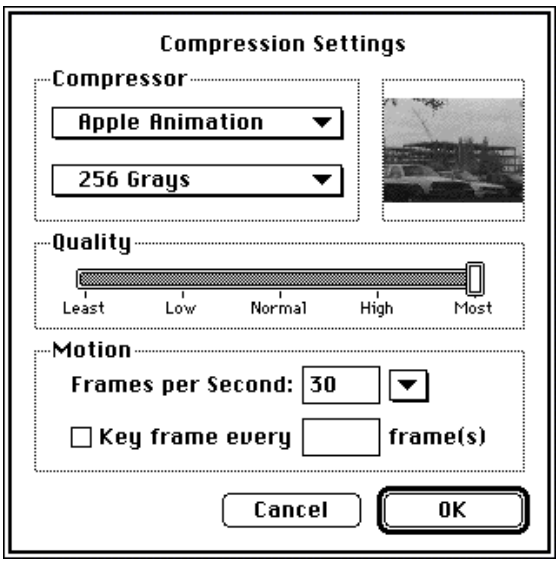


Figure 1-7 shows a dialog box that this application provides to allow the user to select compression methods for video using the standard image-compression dialog component.

Figure 1-7 Compression settings



The remainder of this book provides the technical reference you need to develop an application that lets users display, edit, cut, copy, and paste movies and movie data in the same way that they currently manipulate text and graphic elements.

Chapter 2 discusses the Movie Toolbox, the set of functions with which you can create and modify movies and movie files.

Chapter 3 describes the Image Compression Manager, with which your application can compress and decompress still images and video sequences.

Chapter 4 describes the format and content of movie resources and movie files. This chapter is of interest only to developers of QuickTime components.

The book concludes with a glossary and an index.

CHAPTER 2

Movie Toolbox

Contents

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

This chapter is divided into the following major sections:

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

Introduction to Movies

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

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

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

Time and the Movie Toolbox

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

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

The following sections discuss each of these key concepts further.

Time Coordinate Systems

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

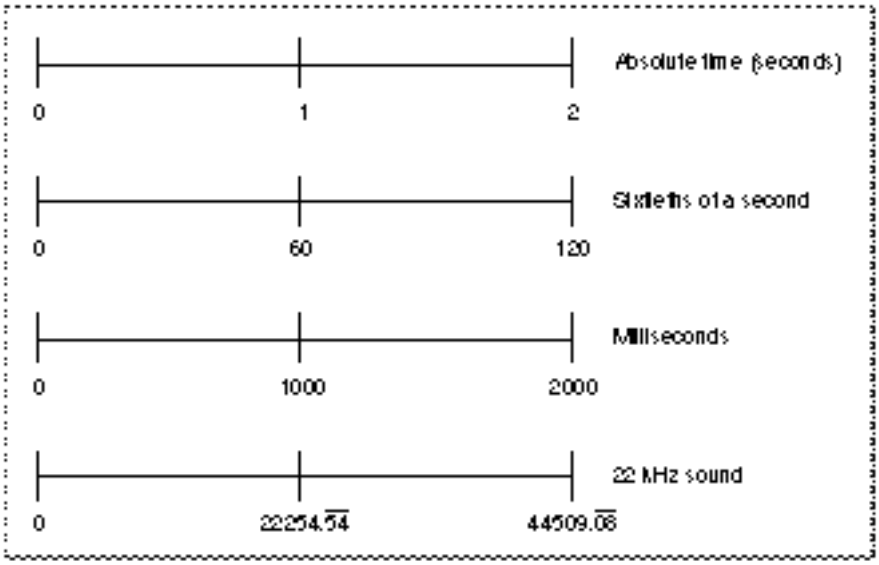
A QuickTime time scale defines the number of time units that pass each second in a given time coordinate system. A time coordinate system that has a time scale of 1 measures time in seconds. Similarly, a time coordinate system that has a time scale of 60 measures sixtieths of a second. In general, each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

Table 2-1 Common movie time scales

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

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

Figure 2-1 Time scales



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

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

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

Time Bases

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

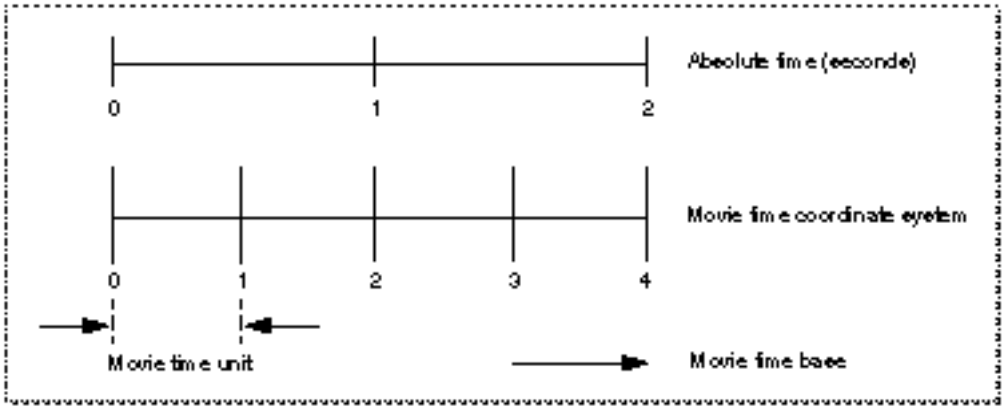
Time bases exist independently of any specific time coordinate system. However, time values extracted from a time base are meaningless without a time scale. Therefore, whenever you obtain a time value from a time base, you must specify the time scale of the time value result. The Movie Toolbox translates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

Note

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

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

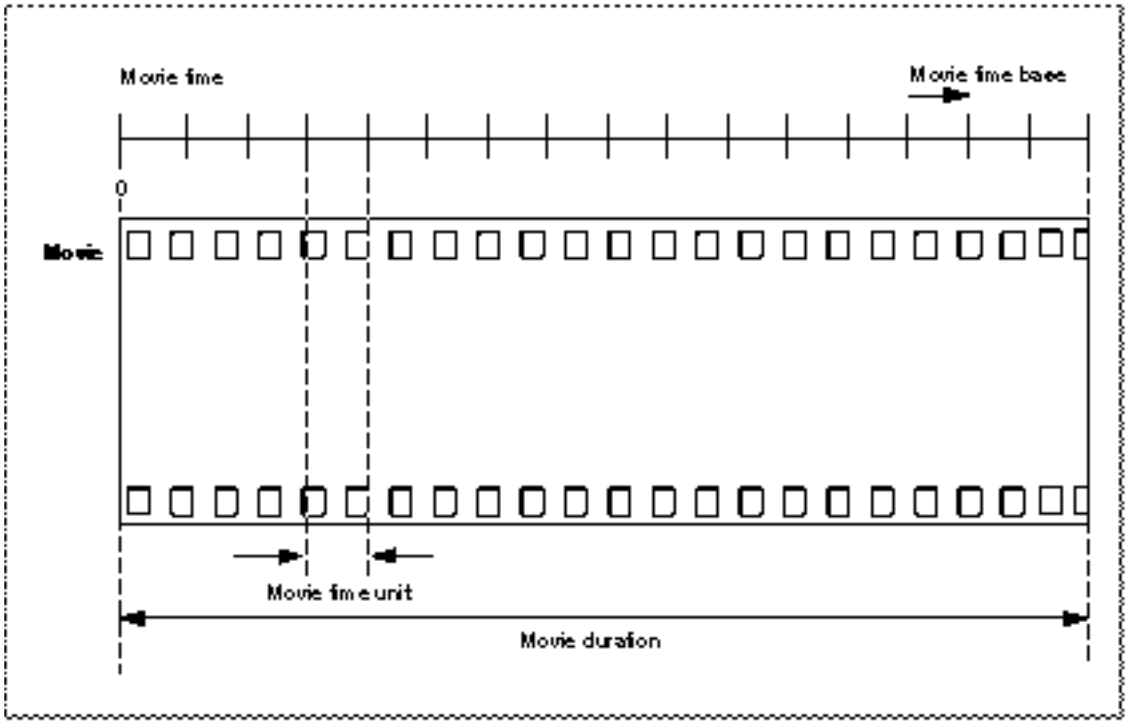
Figure 2-2 A time coordinate system and a time base



Movies

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

Figure 2-3 A movie's time coordinate system



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

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

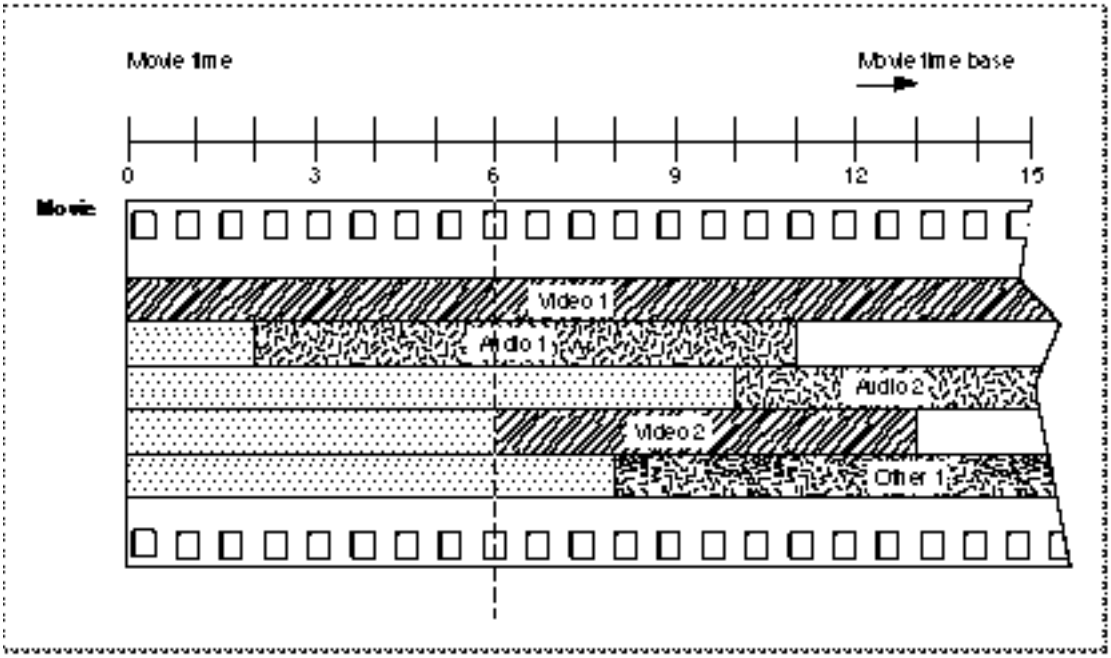
Note

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

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

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

Figure 2-4 A movie containing several tracks

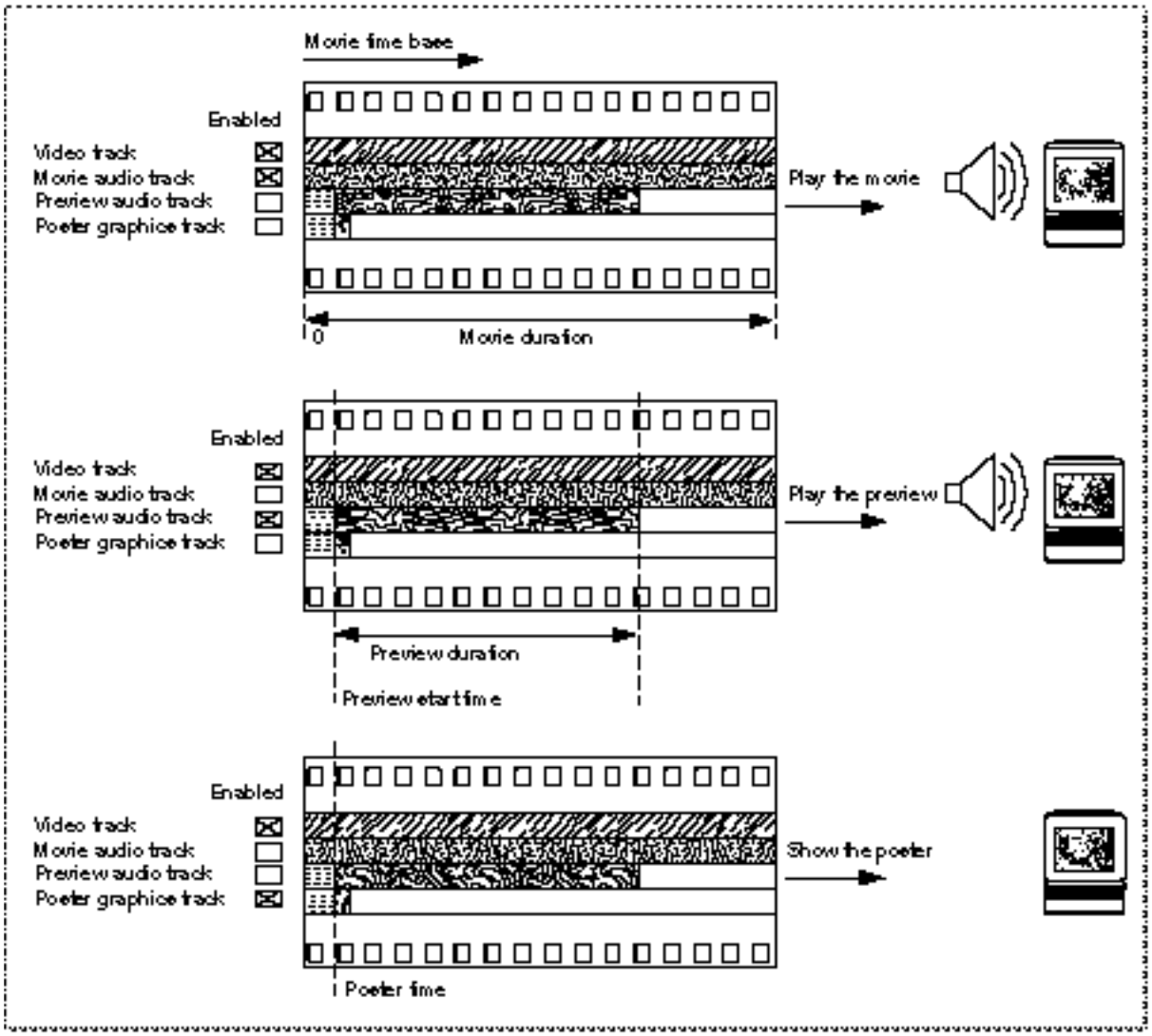


The Movie Toolbox allows you to define both a movie preview and a movie poster for a QuickTime movie. A **movie preview** is a short dynamic representation of a movie. Movie previews typically last no more than 3 to 5 seconds, and they should give the user some idea of what the movie contains. (An example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

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

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

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

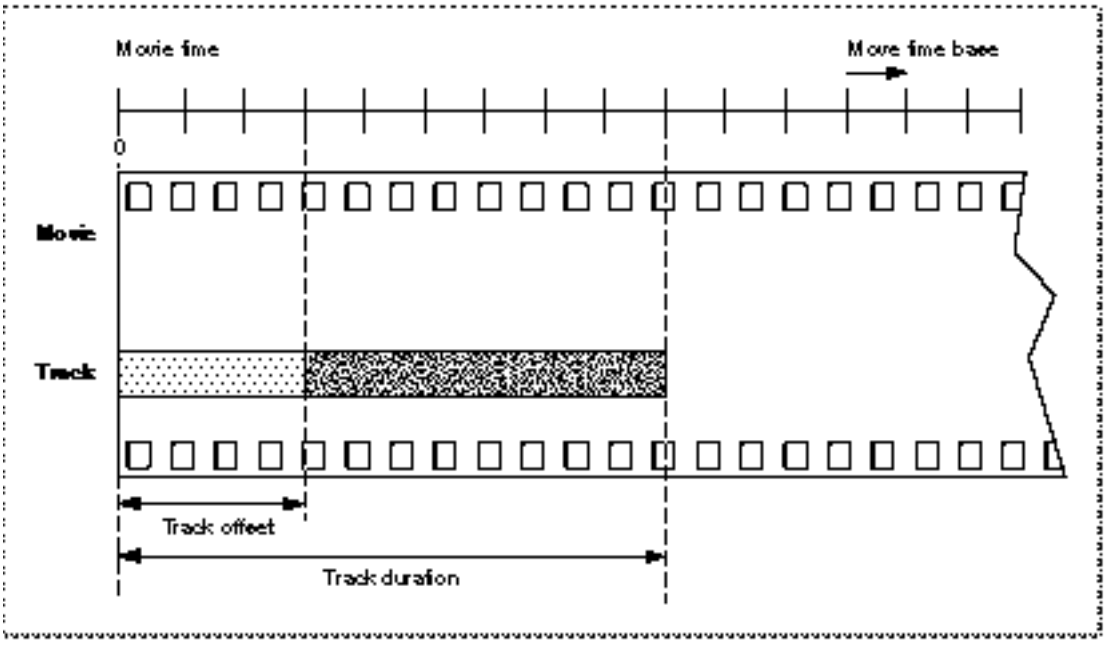


Tracks

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

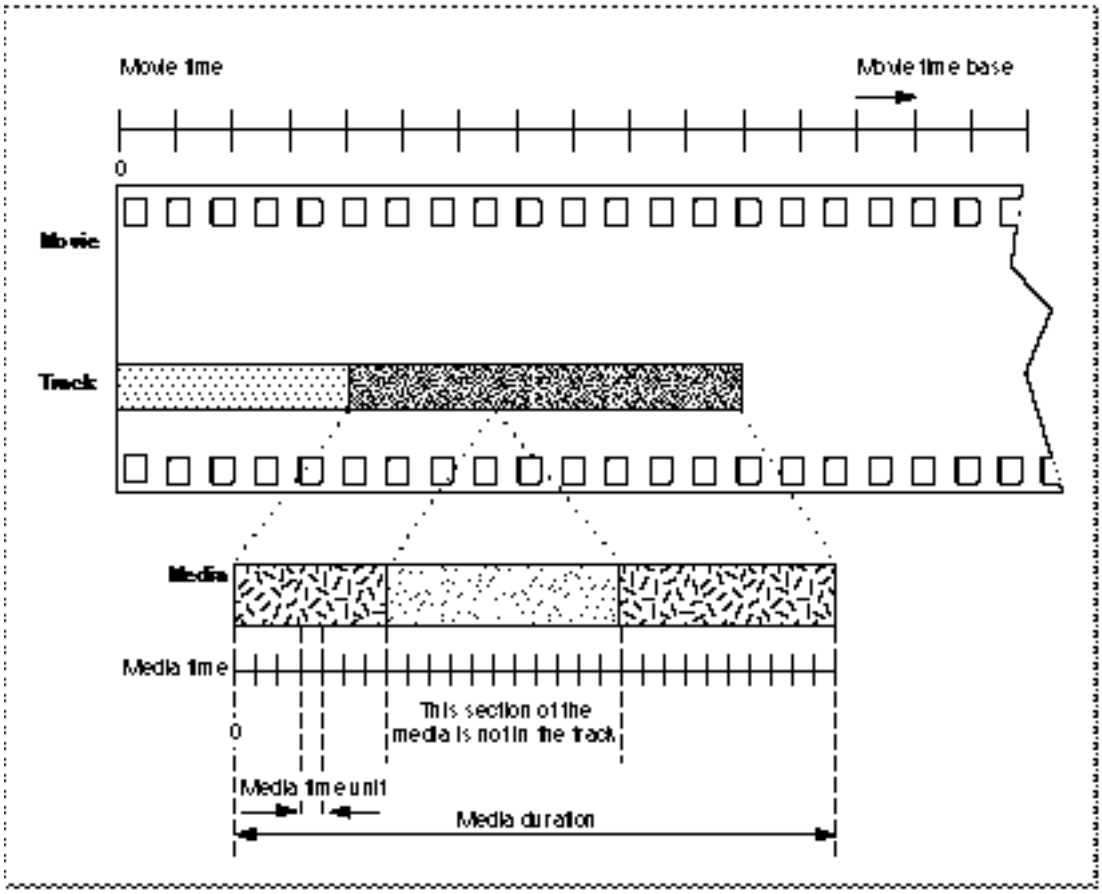
All of the tracks in a movie use the movie's time coordinate system. That is, the movie's time scale defines the basic time unit for each of the movie's tracks. Each track begins at the beginning of the movie, but the track's data might not begin until some time value other than 0. This intervening time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

Figure 2-6 A track in a movie



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

Figure 2-7 A track and its media



Media Structures

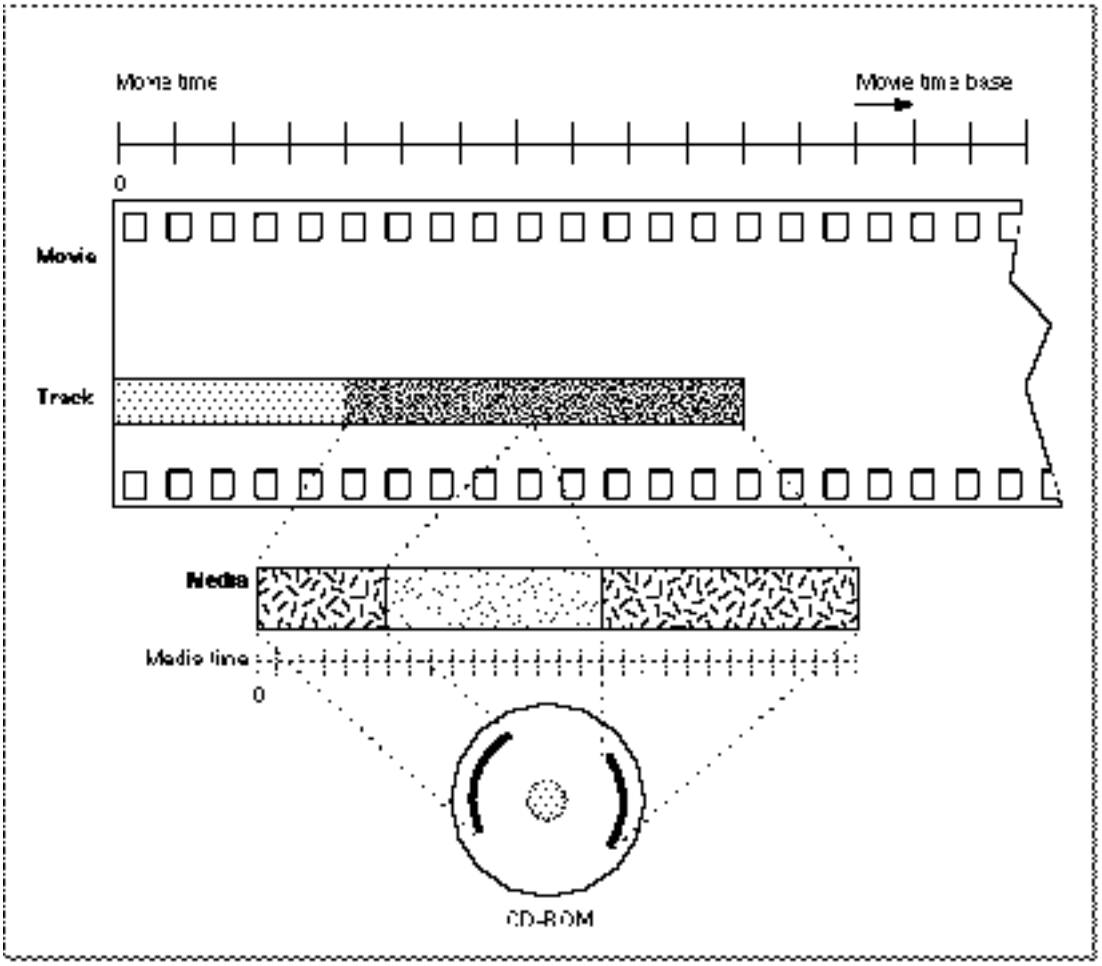
A media describes the data for a track. The data is not actually stored in the media. Rather, the media contains references to its media data, which may be stored in disk files, on CD-ROM discs, or other appropriate storage devices. Note that the data referred to by one media may be used by more than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

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

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

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

Figure 2-8 A media and its data



About Movies

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

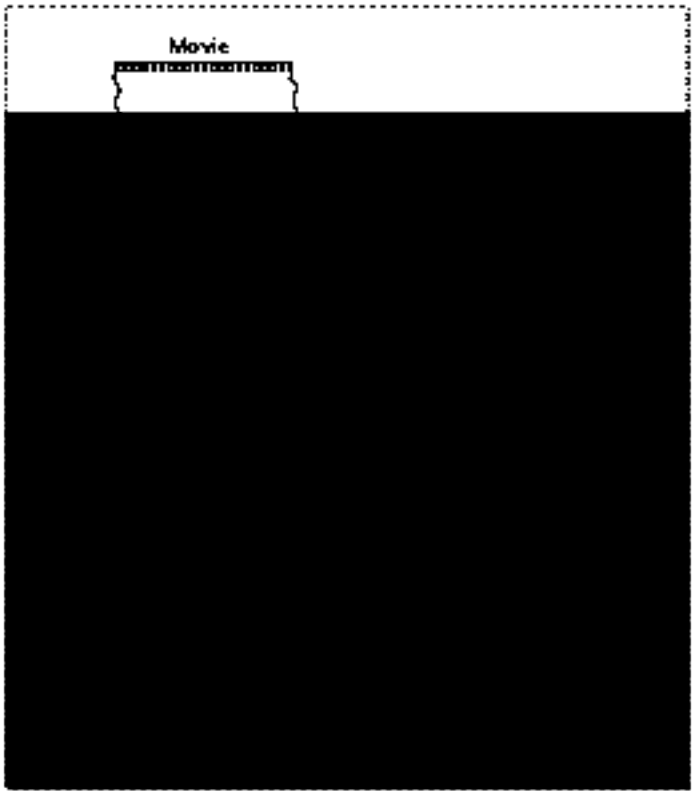
- "Movie Characteristics" discusses the time, display, and sound characteristics of a QuickTime movie
- "Track Characteristics" describes the characteristics of a movie track

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

Movie Characteristics

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

Figure 2-9 Movie characteristics



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

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

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

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

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

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

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

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

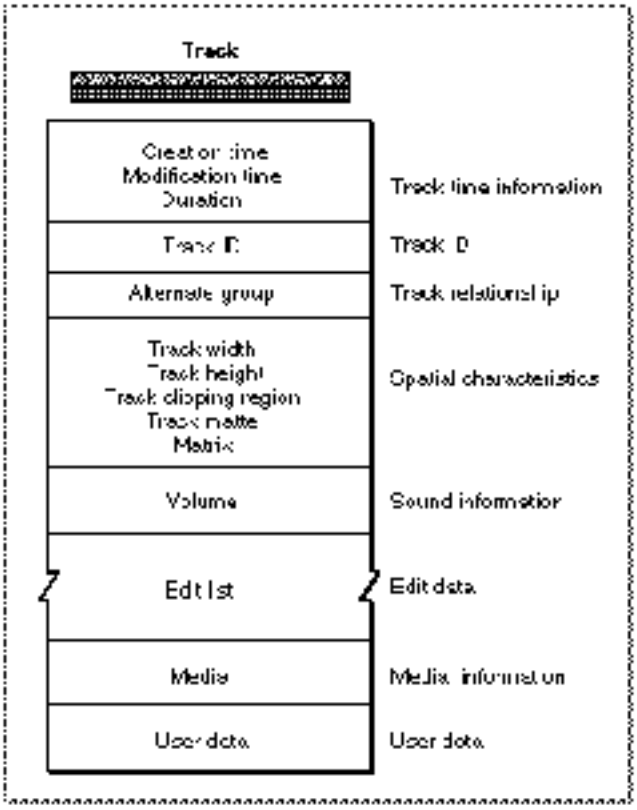
Movies contain each of their tracks. See the next section for more information about tracks and their characteristics.

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

Track Characteristics

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

Figure 2-10 Track characteristics



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

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

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

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

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

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

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

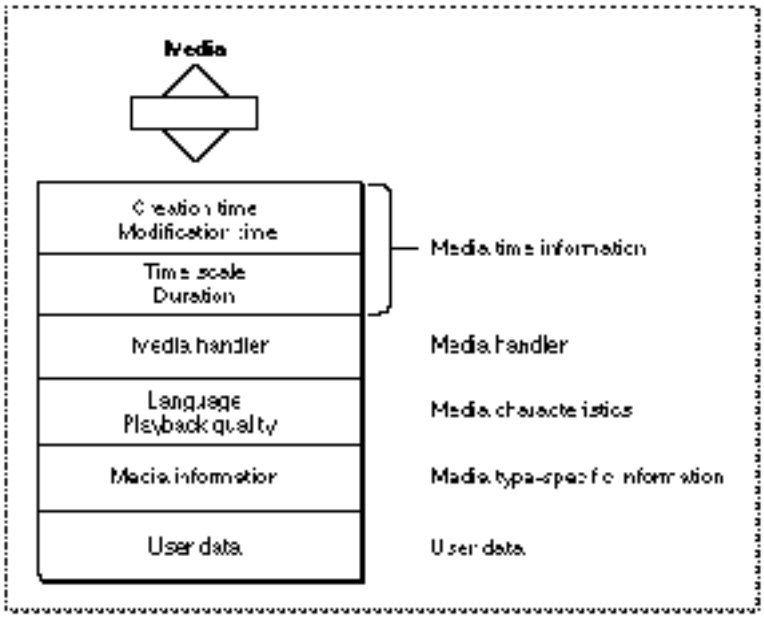
Each track contains its associated media. See the next section for more information about media structures and their characteristics.

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

Media Characteristics

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

Figure 2-11 Media characteristics



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

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

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

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

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

Spatial Properties

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

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

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

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

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

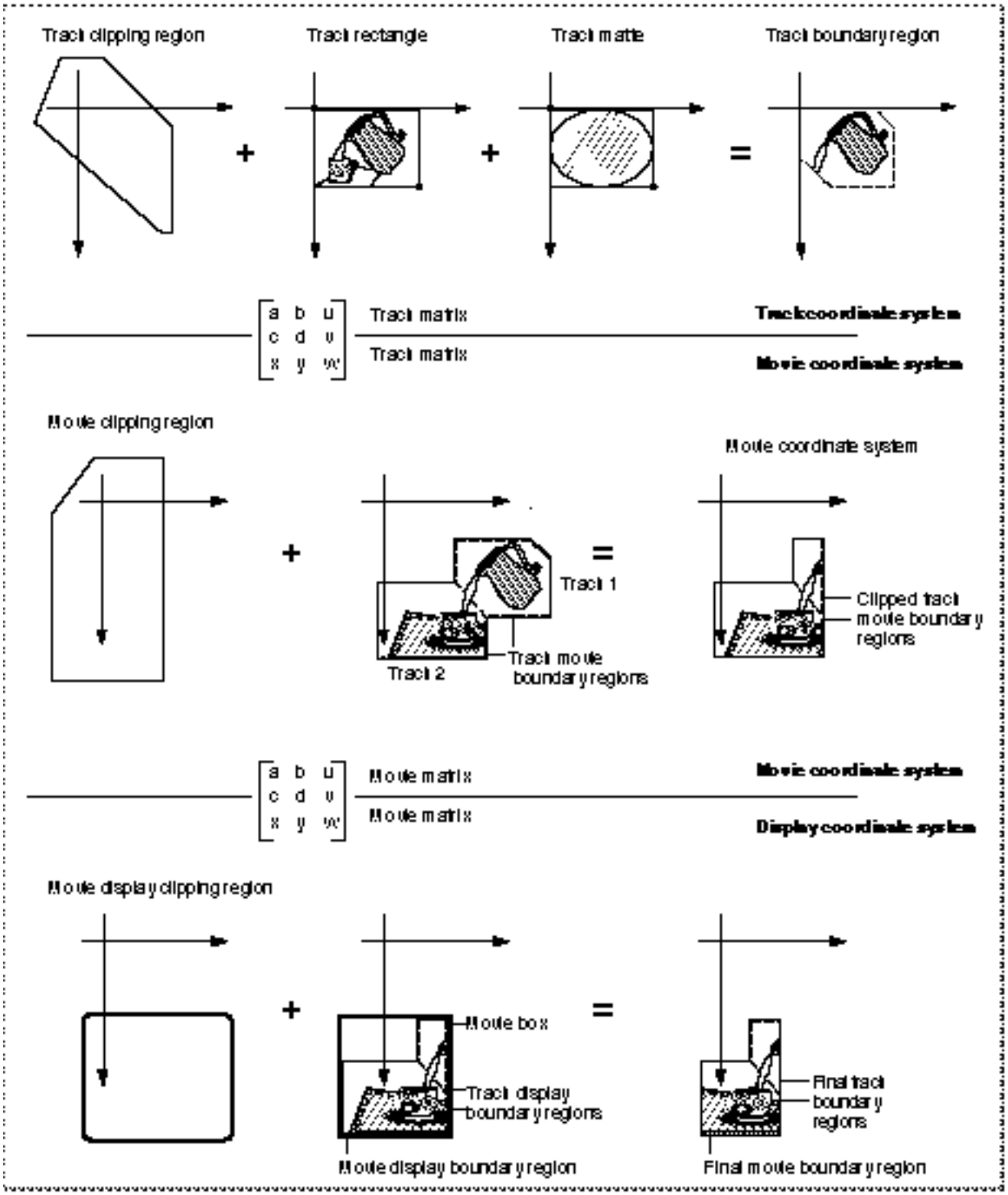
This section describes the process that the Movie Toolbox uses to display a movie. The process begins with the movie data and ends with the final movie display. The phases, which are described in detail in this section, include

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

Note

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

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

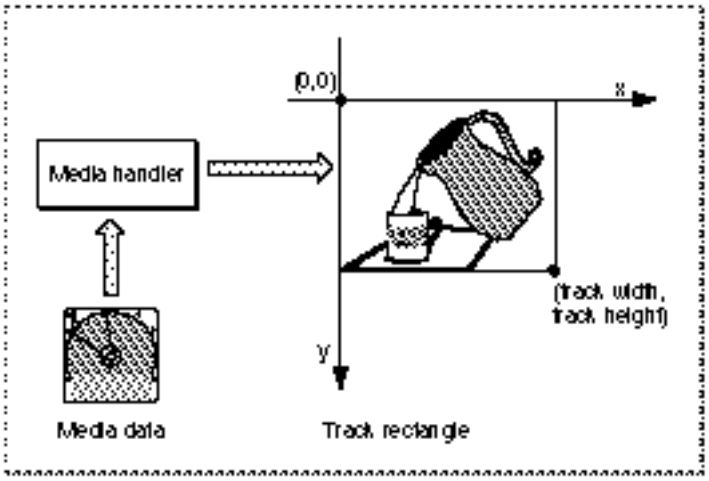


Each track defines a rectangle into which its media is displayed. This rectangle is referred to as the **track rectangle,** and it is defined by the **track width** and **track height** values assigned to the track. The upper-left corner of this rectangle defines the origin point of the track's *coordinate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

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

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

Figure 2-13 A track rectangle



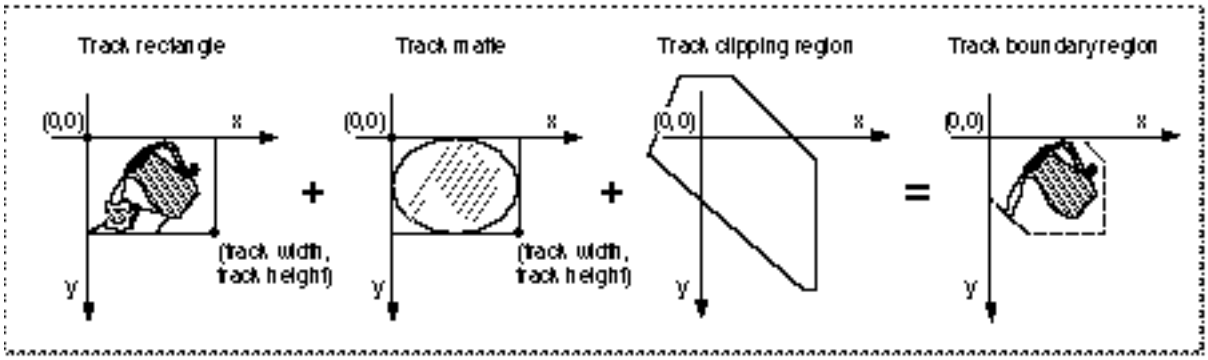
The Movie Toolbox next mattes the image in the track rectangle by applying the track matte and the track clipping region. This does not affect the shape of the image—only the display. Both the track matte and the track clipping region are optional.

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

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

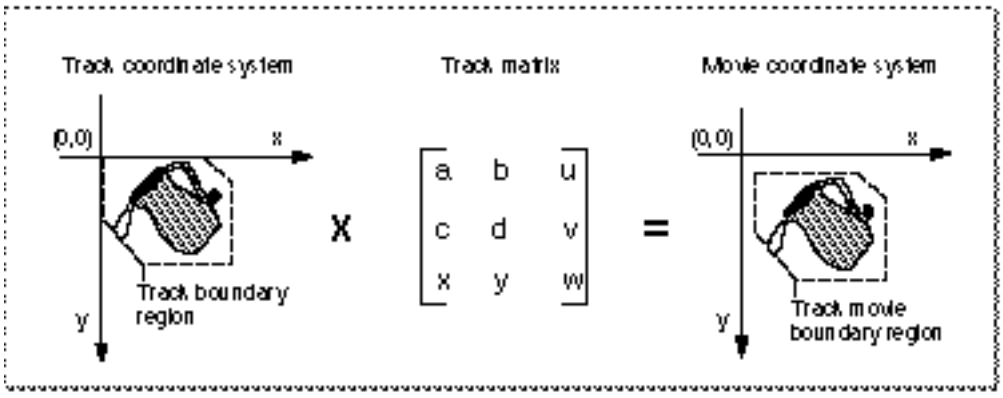
This process and its results are shown in Figure 2-14.

Figure 2-14 Clipping a track's image



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

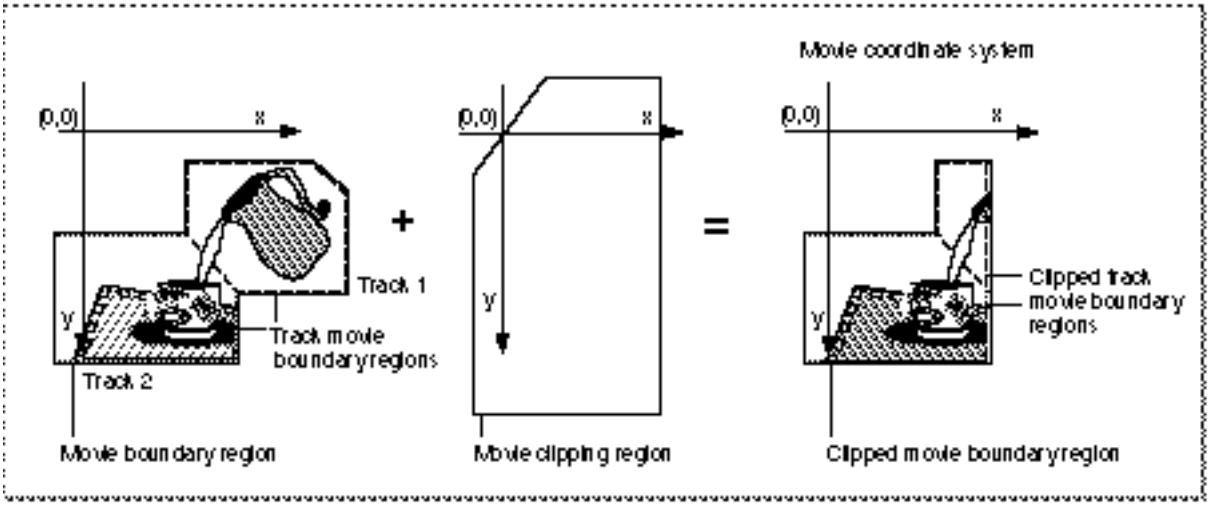
Figure 2-15 A track transformed into a movie coordinate system



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

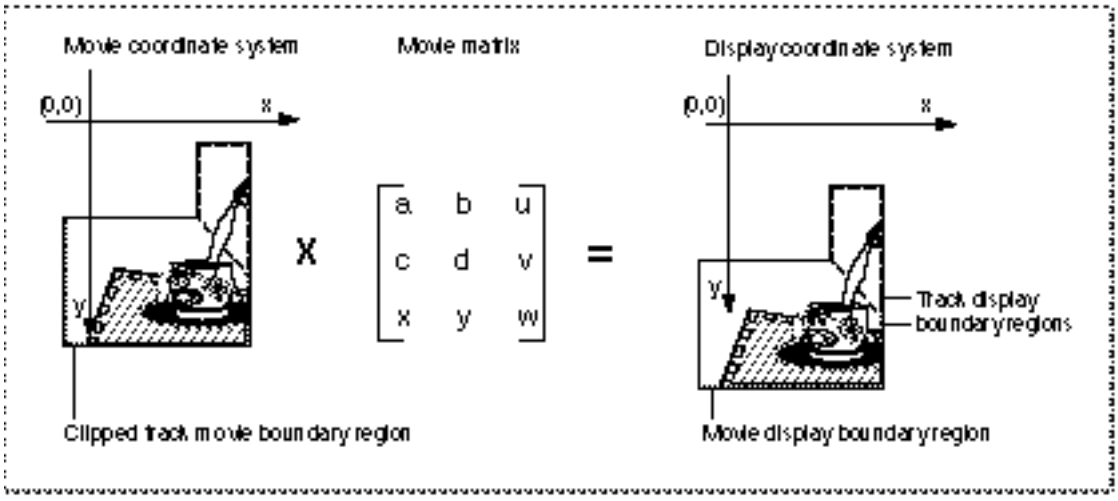
The union of all track movie boundary regions for a movie defines the movie's **movie boundary region.** The Movie Toolbox combines a movie's tracks into this single region where layers are applied. Therefore, tracks in back layers may be partially or completely obscured by tracks in front layers. The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

Figure 2-16 Clipping a movie's image



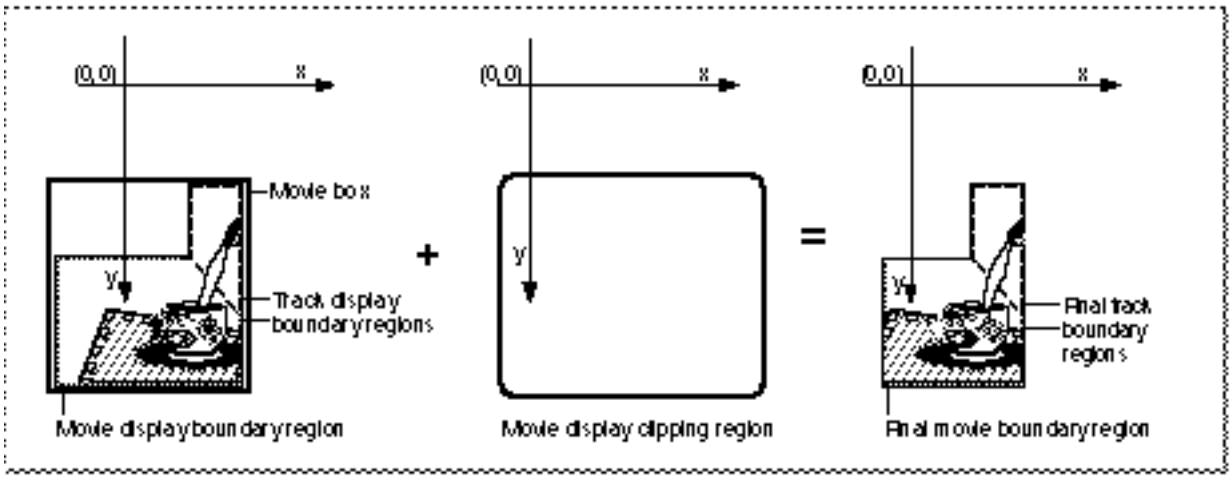
After clipping the movie's image, the Movie Toolbox transforms the resulting image into the display coordinate system. The Movie Toolbox uses a 3-by-3 transformation matrix to accomplish this operation (see the next section, "The Transformation Matrix," for a complete discussion of matrix operations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

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



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

Figure 2-18 Clipping a movie for final display



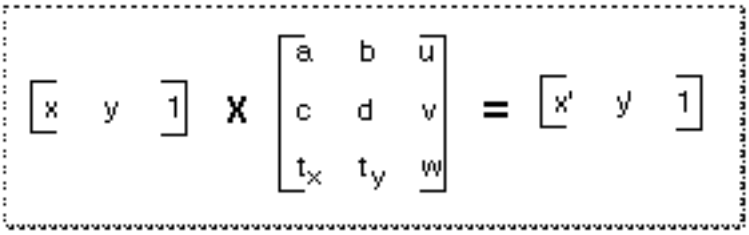
Once the movie is in the **display coordinate system** (that is, the QuickDraw graphics world), the Movie Toolbox performs a final clipping operation to generate the image that is displayed. The movie is clipped with the **movie display clipping region.** When a movie is displayed, the Movie Toolbox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

The Transformation Matrix

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

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

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



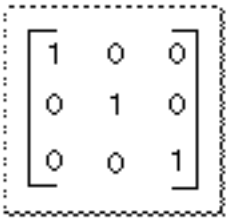
During display operations, the contents of a 3-by-3 matrix transform a point (x,y) into a point (x',y') by means of the following equations:

$x' = ax + cy + t_x$

$y' = bx + dy + t_y$

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

Figure 2-20 The identity matrix



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):

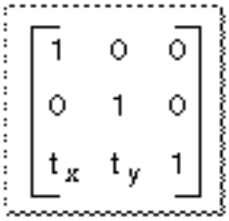
x' = 1x + 0y + 0

y' = 0x + 1y + 0

x' = y and y' = y

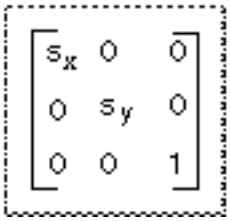
In order to move an image by a specified displacement, you perform a translation operation. This operation modifies the x and y coordinates of each point by a specified amount. The matrix shown in Figure 2-21 describes a translation operation.

Figure 2-21 A matrix that describes a translation operation



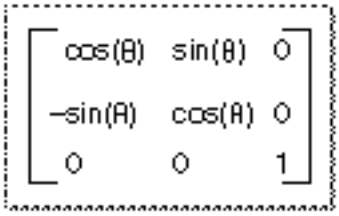
You can stretch or shrink an image by performing a scaling operation. This operation modifies the x and y coordinates by some factor. The magnitude of the x and y factors governs whether the new image is larger or smaller than the original. In addition, by making the x factor negative, you can flip the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

Figure 2-22 A matrix that describes a scaling operation



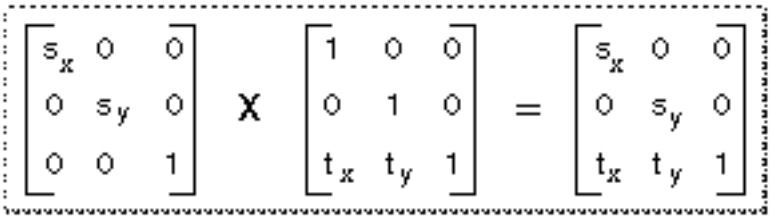
Finally, you can rotate an image by a specified angle by performing a rotation operation. You specify the magnitude and direction of the rotation by specifying factors for both x and y. The matrix shown in Figure 2-23 rotates an image counterclockwise by an angle θ.

Figure 2-23 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 2-24.

Figure 2-24 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. For more information about the `Fract` data type, see *Inside Macintosh: Imaging*.

Audio Properties

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

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

Sound Playback

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

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

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

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

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

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

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

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

Adding Sound to Video

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

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

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

Sound Data Formats

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

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

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

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

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

Sample rate indicates the number of samples captured per second. The sample rate also influences the sound quality, because higher rates can more accurately capture the original sound waveform. The basic Macintosh hardware supports an output sampling rate of 22.2$\overline{54}$ kHz. The Movie Toolbox can support any rate up to 65.535 kHz; as with sample size, the Movie Toolbox converts higher sample rates to rates that can be accommodated by the Macintosh hardware when it plays the sound.

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

Data Interchange

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

Movies on the Clipboard

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

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

Movies in Files

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

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

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

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

Using the Movie Toolbox

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

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

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

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

Determining Whether the Movie Toolbox Is Installed

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

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

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

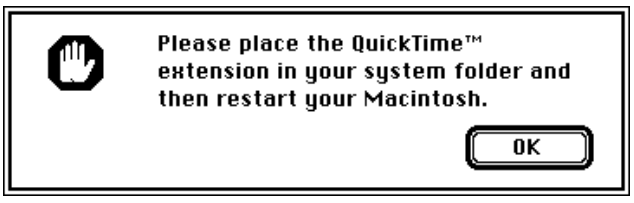
The code in Listing 2-1 contains a function that demonstrates how your application can call the Gestalt Manager.

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

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

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

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

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

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

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



Getting Ready to Work With Movies

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

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

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

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

Getting a Movie From a File

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

Listing 2-2 Getting a movie from a file

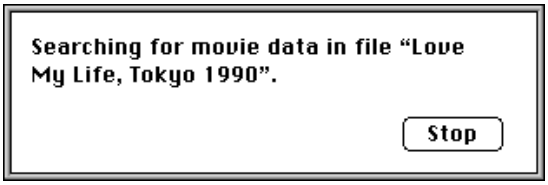
```
Movie GetMovie (void)
{
   OSErr             err;
   SFTypeList        typeList = {MovieFileType,0,0,0};
   StandardFileReply reply;
   Movie             aMovie = nil;
   short             movieResFile;

   StandardGetFilePreview (nil, 1, typeList, &reply);
   if (reply.sfGood)
   {
      err = OpenMovieFile (&reply.sfFile, &movieResFile,
                           fsRdPerm);
      if (err == noErr)
      {
         short       movieResID = 0;   /* want first movie */
```

```
        Str255      movieName;
        Boolean     wasChanged;

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

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

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

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



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

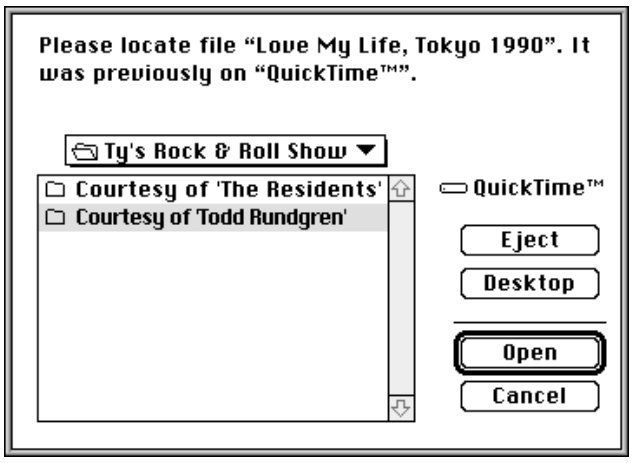
Figure 2-27 A dialog box that informs the user the movie file cannot be found

The user has two options:

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

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

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



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

Figure 2-29 An alert for an invalid movie file

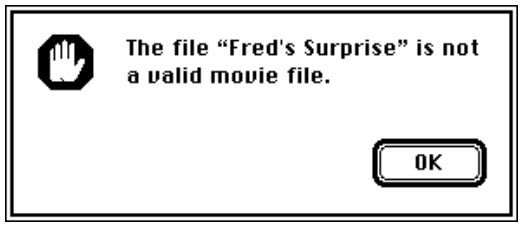
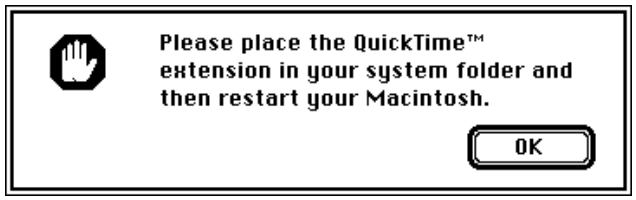


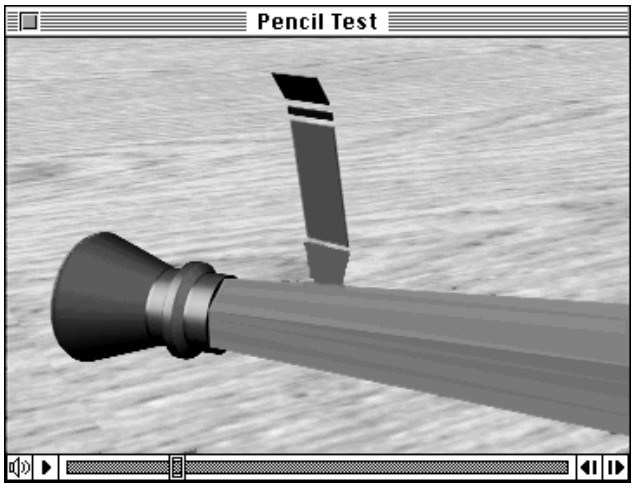
Figure 2-30 An alert when QuickTime cannot be found



Playing Movies With a Movie Controller

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

Figure 2-31 A movie controller playing a movie



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

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

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

void main (void)
{
```

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

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

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

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

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

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

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

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

Playing a Movie

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

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

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

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

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

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

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

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

Listing 2-4 Playing a movie

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

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

/* #include "QuickTimeComponents.h" */

#define doTheRightThing 5000

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

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

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

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

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

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

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

StartMovie (aMovie);

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

         case mouseDown:
            part = FindWindow (theEvent.where,
                               &whichWindow);
            if (whichWindow == aWindow)
            {
               switch (part)
               {
                  case inGoAway:
                     done = TrackGoAway (whichWindow,
                                         theEvent.where);
                     break;
                  case inDrag:
                     DragWindow (whichWindow,
                                 theEvent.where,
                                 &qd.screenBits.bounds);
                     break;
               }
            }
            break;
      }
   }
```

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

Movies and the Scrap

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

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

Creating a Movie

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

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

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

Note

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

A Sample Program for Creating a Movie

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

Listing 2-5 Creating a movie: The main program

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

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


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

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

void InitMovieToolbox (void)
{
      OSErr err;

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

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

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

A Sample Function for Creating and Opening a Movie File

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

Listing 2-6 Creating and opening a movie file

```
#define kMyCreatorType 'TVOD'

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

#define kPrompt "\pEnter movie file name:"

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

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

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

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

    CreateMyVideoTrack (theMovie);
    CreateMySoundTrack (theMovie);

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

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

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

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

Listing 2-7 Creating a video track

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

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

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

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

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

   AddVideoSamplesToMedia (theMedia, &trackFrame);

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

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

A Sample Function for Adding Video Samples to a Media

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

Listing 2-8 Adding video samples to a media

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

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

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

   LockPixels (theGWorld->portPixMap);
```

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

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

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

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

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

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

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

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

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

   SetGWorld (oldPort, oldGDeviceH);

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

A Sample Function for Creating Video Data for a Movie

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

Listing 2-9 Creating video data

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

   ForeColor( redColor );
   PaintRect( trackFrame );

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

A Sample Function for Creating a Sound Track

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

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

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

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

Listing 2-10 Creating a sound track

```
#define  kSoundSampleDuration  1
#define  kSyncSample 0
#define  kTrackStart 0
#define  kMediaStart 0
#define  kFix1       0x00010000


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


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

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

   CreateSoundDescription (sndHandle,
                    sndDesc,
```

```
                    &sndDataOffset,
                    &numSamples,
                    &sndDataSize );

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

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

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

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

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

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

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

A Sample Function for Creating a Sound Description Structure

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

Listing 2-11 Creating a sound description

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

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

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

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

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

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

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

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

   case extSH:
   {
      ExtSoundHeaderPtr    extSndHdrP;

      extSndHdrP = (ExtSoundHeaderPtr)sndHdrPtr;
```

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

   case cmpSH:
   {
      CmpSoundHeaderPtr cmpSndHdrP;

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

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

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

```
            break;

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

Parsing a Sound Resource

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

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

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

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

Listing 2-12 Parsing a sound resource

```
typedef SndCommand *SndCmdPtr;

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

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

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


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

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

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

   howManyCmds = *(short *)sndPtr;

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

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

   return sndOffset;
}     /* of GetSndHdrOffset */
```

Saving Movies in Movie Files

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

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

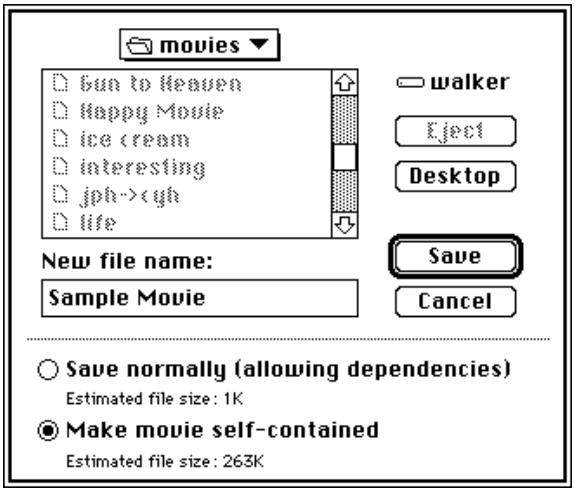
If you are working with an existing movie, you must read that movie from a movie file or choose a movie from the scrap. You first open the movie file by calling the `OpenMovieFile` function (described on page 2-98). You then load the movie from that file by calling the `NewMovieFromFile` function (described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

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

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

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

Figure 2-32 A sample movie Save As dialog box



Using Movies in Your Event Loop

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

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

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

Note

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

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

Listing 2-13 Handling movie update events

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

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

The Movie Toolbox and System 6

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

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

Note

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

The Alias Manager

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

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

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

Note

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

The File Manager

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

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

```
FSMakeFSSpec      FSpOpenDF
FSpOpenRF         FSpCreate
FSpDirCreate      FSpDelete
FSpGetFInfo       FSpSetFInfo
FSpSetFLock       FSpRstFLock
```

```
FSpRename          FSpCatMove
FSpOpenResFile     FSpCreateResFile
FSpGetCatInfo
```

QuickTime does not support the `FSpExchangeFiles` function.

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

Previewing Files

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

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

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

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

Previewing Files in System 6 Using Standard File Reply Structures

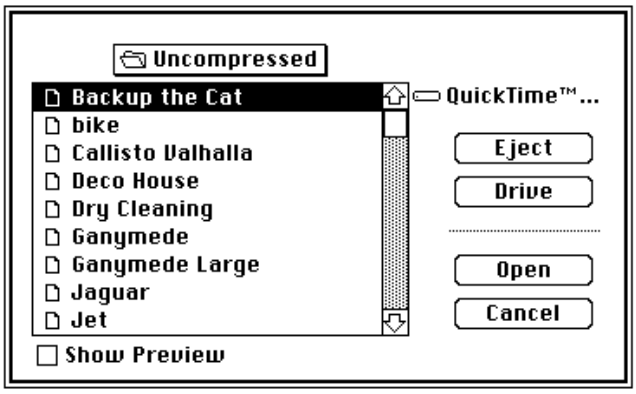
The Movie Toolbox provides two new standard file functions that allow you to display file previews in an Open dialog box in System 6 using standard file reply structures: `SFGetFilePreview` and `SFPGetFilePreview`. The `SFGetFilePreview` function (described on page 2-306) corresponds to the existing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

IMPORTANT

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

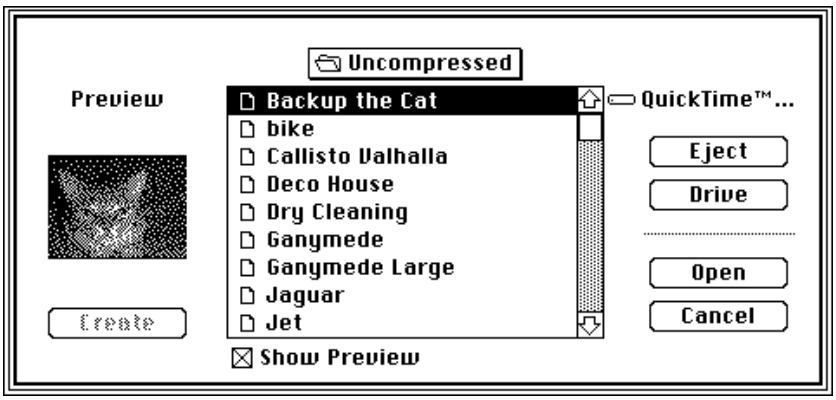
The `SFGetFilePreview` function uses the dialog box shown in Figure 2-33. The `SFPGetFilePreview` function can also use this dialog box, if you do not supply your own.

Figure 2-33 `SFGetFilePreview` or `SFPGetFilePreview` dialog box without preview



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

Figure 2-34 `SFGetFilePreview` or `SFPGetFilePreview` dialog box with preview



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

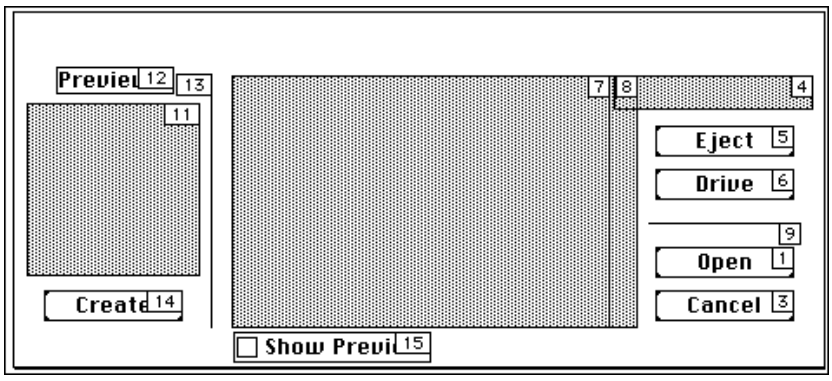
Customizing Your Interface in System 6

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

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

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

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



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

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

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

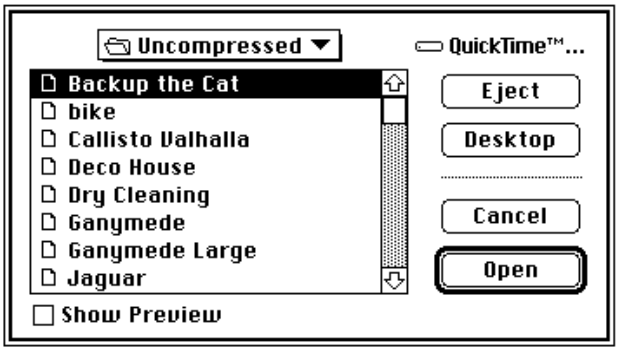
```
   sfpItemPreviewDividerUser  = 13,    /* user divider preview */
   sfpItemCreatePreviewButton = 14,    /* create preview button */
   sfpItemShowPreviewButton   = 15     /* show preview button */
};
```

Previewing Files in System 7 Using Standard File Reply Structures

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

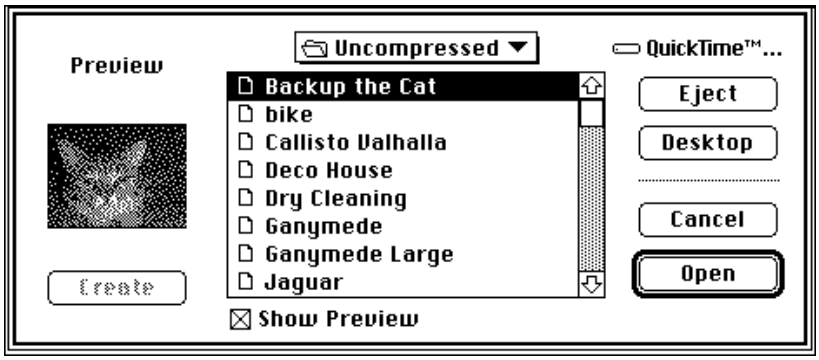
The `StandardGetFilePreview` function uses the dialog box shown in Figure 2-36. The `CustomGetFilePreview` function can also use this dialog box, if you do not supply your own.

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



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

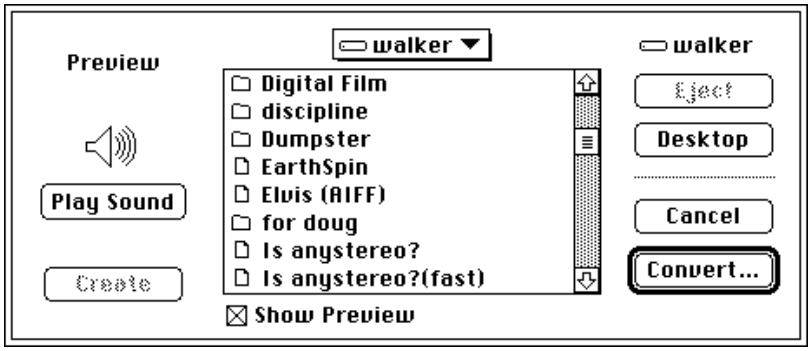
Figure 2-37 `StandardGetFilePreview` or `CustomGetFilePreview` dialog box with preview



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

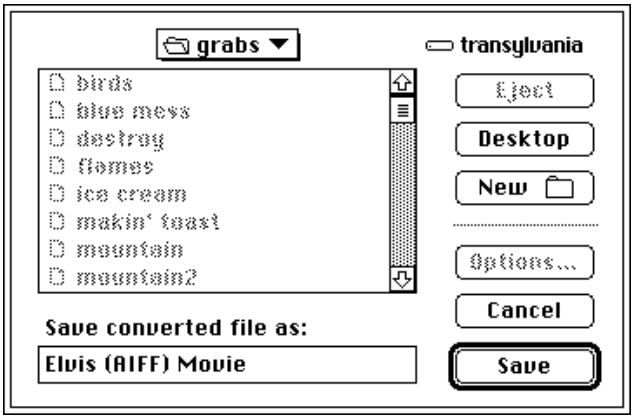
The `SFGetFilePreview`, `SFPGetFilePreview`, `StandardGetFilePreview`, and `CustomGetFilePreview` functions allow the user to automatically convert files to movies if your application requests movies. If there is a file that can be converted into a movie file using a movie import component, then the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

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



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

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



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

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

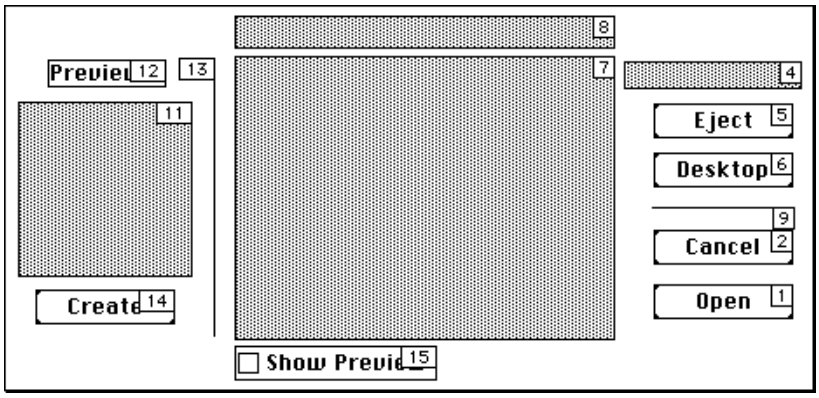
Customizing Your Interface in System 7

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

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

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

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



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

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

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

Using Application-Defined Functions

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

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

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

Listing 2-14 Two sample movie cover functions

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

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

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

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

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

   SetGWorld (curPort, curGD);
   return noErr;
}

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

```
   RgnHandle   displayBounds;
   GrafPtr     curPort;

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

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

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

Working With Movie Spatial Characteristics

The following section provides an example of how to create a track matte.

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

Listing 2-15 Creating a track matte

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

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

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

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

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

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

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

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

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

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

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

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

   DisposeMatte (trackMatte);
}
```

Movie Toolbox Reference

This section describes all the Movie Toolbox data types and functions. The Movie Toolbox provides a rich and varied set of functions that allow your application to work with QuickTime movies. This discussion has been divided into the following sections:

- "Data Types" identifies the data types used by your application when interacting with the Movie Toolbox
- "Functions for Getting and Playing Movies" describes the functions that applications can use to create, get, and play movies
- "Functions That Modify Movie Properties" describes functions that allow you to change the display, time, and sound characteristics of a movie
- "Functions for Editing Movies" discusses the functions that you can use to edit the contents of movies
- "Media Functions" discusses the functions that allow you to communicate with media handlers
- "Functions for Creating File Previews" describes the functions provided by the Movie Toolbox that allow you to create file previews
- "Functions for Displaying File Previews" describes the Movie Toolbox functions that let you display file previews
- "Time Base Functions" discusses the various Movie Toolbox functions that work with time bases
- "Matrix Functions" describes the Movie Toolbox functions that allow you to manipulate transformation matrices
- "Application-Defined Functions" describes the functions your application can provide when interacting with the Movie Toolbox

If you are developing a QuickTime-aware application that plays existing movies, you should read "Functions for Getting and Playing Movies," which begins on page 2-81.

If you are developing an application that allows the user to create and edit movies, you should also read "Functions for Editing Movies," which begins on page 2-242. More advanced display and editing applications may use some of the functions described in "Functions That Modify Movie Properties," which begins on page 2-157.

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.

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:

`Media` Specifies the media for an operation. Your application obtains a media identifier from such Movie Toolbox functions as `NewTrackMedia` and `GetTrackMedia` (described on page 2-153 and page 2-206, respectively).

`Movie` Specifies the movie for an operation. Your application obtains a movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

`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 (described on page 2-255).

`QTCallBack` Specifies the callback for an operation. You obtain a callback identifier from the `NewCallBack` function (described on page 2-336).

`TimeBase` Specifies the time base for an operation. Your application obtains a time base identifier from the `NewTimeBase` or `GetMovieTimeBase` functions (described on page 2-316 and page 2-190, respectively).

`Track` Specifies the track for an operation. Your application obtains a track identifier from such Movie Toolbox functions as `NewMovieTrack` and `GetMovieTrack` (described on page 2-151 and page 2-204, respectively).

`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 (described on page 2-269).

`UserData` Specifies the user data list for an operation. You obtain a user data list identifier by calling the `GetMovieUserData`, `GetTrackUserData`, or `GetMediaUserData` functions (described on page 2-231, page 2-232, and page 2-233, respectively).

The Time Structure

The Movie Toolbox provides a number of functions that allow you to work with time specifications. These functions are described in "Time Base Functions" beginning on page 2-315. 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 descriptions

`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 (described on page 2-190 and page 2-316, respectively).

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 descriptions

`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 (described on page 2-349), 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 descriptions

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 descriptions

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.

The Sound Description Structure

A sound description structure contains information that defines the characteristics of one or more sound samples. Data in the sound description structure indicates the type of compression that was used, the sample size, the rate at which samples were obtained, and so on. Sound media handlers use the information in the sound description structure when they process the sound samples.

See the chapter "Image Compression Manager" for a description of the image description structure, which contains information that defines the characteristics of an image.

The `SoundDescription` data type defines the layout of a sound description structure. See "Media Functions," which begins on page 2-281, for more information about sound media handlers.

```
struct SoundDescription
{
   long  descSize;      /* number of bytes in this structure */
   long  dataFormat;    /* format of the sound data */
   long  resvd1;        /* reserved--set to 0 */
   short resvd2;        /* reserved--set to 0 */
   short dataRefIndex;  /* reserved--set to 1 */
   short version;       /* reserved--set to 0 */
   short revlevel;      /* reserved--set to 0 */
   long  vendor;        /* reserved--set to 0 */
   short numChannels;   /* number of channels used by sample */
   short sampleSize;    /* number of bits in each sample */
   short compressionID; /* reserved--set to 0 */
   short packetSize;    /* reserved--set to 0 */
   Fixed sampleRate;    /* rate at which samples were obtained */
};
```

Field descriptions

`descSize` Defines the total size, in bytes, of this sound description structure.

`dataFormat` Describes the format of the sound data. Possible values include:

`'raw '`	Sound samples are stored uncompressed, in offset-binary format (that is, sample data values range from 0 to 255).
`'twos'`	Sound samples are stored uncompressed, in twos-complement format (that is, sample data values range from –128 to 127). The Sound Manager uses this format when it creates sound files in Audio Interchange File Format (AIFF).
`'MAC3'`	Sound samples have been compressed by the Sound Manager at a ratio of 3:1.
`'MAC6'`	Sound samples have been compressed by the Sound Manager at a ratio of 6:1.

Some older movie files sometimes have a zero value in this field. You should assume that this is the same as the `'raw '` value.

`resvd1` Reserved for Apple. Set this field to 0 in any sound description structures you create.

`resvd2` Reserved for Apple. Set this field to 0 in any sound description structures you create.

dataRefIndex	Reserved for Apple. Set this field to 0 in any sound description structures you create.
version	Reserved for Apple. Set this field to 0 in any sound description structures you create.
revLevel	Reserved for Apple. Set this field to 0 in any sound description structures you create.
vendor	Reserved for Apple. Set this field to 0 in any sound description structures you create.
numChannels	Indicates the number of sound channels used by the sound sample. Set this field to 1 for monaural sounds; set it to 2 for stereo sounds.
sampleSize	Specifies the number of bits in each sound sample. Set this field to 8 for 8-bit sound; set it to 16 for 16-bit sound.
compressionID	Reserved for Apple. Set this field to 0 in any sound description structures you create.
packetSize	Reserved for Apple. Set this field to 0 in any sound description structures you create.
sampleRate	Indicates the rate at which the sound samples were obtained. Sound media handlers use this value to influence the natural playback speed of the sound described by this sound description structure. This field contains an unsigned, fixed-point number that specifies the number of samples collected per second. Some common values include:

0x15BBA2E8	Specifies a sample rate of 5563.6363 samples per second.
0x1CFA2E8B	Specifies a sample rate of 7418.1818 samples per second.
0x2B7745D1	Specifies a sample rate of 11127.2727 samples per second.
0x56EE8BA3	Specifies a sample rate of 22254.5454 samples per second.
0xAC440000	Specifies a sample rate of 44100.0000 samples per second.

Functions for Getting and Playing Movies

The Movie Toolbox provides a number of functions that allow applications to get and play movies. There are also a number of functions that allow you to create new movies. This section describes those functions and has been divided into the following topics:

- "Initializing the Movie Toolbox" discusses the functions that your application must use to gain access to the Movie Toolbox
- "Error Functions" discusses the Movie Toolbox functions that allow you to work with error codes returned by Movie Toolbox functions
- "Movie Functions" describes functions that your application can use to create and access movie resources and movie files

- "Saving Movies" describes the Movie Toolbox functions that allow you to save movies
- "Controlling Movie Playback" describes the functions that you can use to control movie playback
- "Movie Posters and Movie Previews" discusses the functions that allow applications to work with movie posters and movie previews
- "Movies and Your Event Loop" discusses the Movie Toolbox functions that your application must call from its main event loop
- "Preferred Movie Settings" describes functions your application can use to set the preferred playback settings of a movie
- "Enhancing Movie Playback Performance" discusses several techniques for improving movie playback performance
- "Disabling Movies and Tracks" describes the functions that allow your application to disable movies and tracks
- "Generating Pictures From Movies" discusses the Movie Toolbox functions that allow your application to create pictures from movie data
- "Creating Tracks and Media Structures" describes the functions your application must use to create new data for a movie
- "Working With Progress and Cover Functions" describes the functions that allow you to specify a custom function that is called during movie playback

Initializing the Movie Toolbox

The Movie Toolbox maintains state information for every application that is currently using the toolbox. The toolbox uses this information to keep track of the application's movies. Before calling any other Movie Toolbox functions, your application must establish this working environment by calling the `EnterMovies` function. When your application is finished with the Movie Toolbox, you can release this storage by calling the `ExitMovies` function.

EnterMovies

Before you call any Movie Toolbox functions, you must initialize the toolbox. Use the `EnterMovies` function to initialize the Movie Toolbox. When your application calls this function, the Movie Toolbox creates its private storage area for your application.

You should initialize any other Macintosh managers your application uses before calling the `EnterMovies` function.

```
pascal OSErr EnterMovies (void);
```

DESCRIPTION

If the `EnterMovies` function fails, it returns an error value—be sure to check the value returned by this function before using any other facilities of the Movie Toolbox.

In addition, you should use the Gestalt Manager to determine whether the Movie Toolbox is installed (see "Determining Whether the Movie Toolbox Is Installed" beginning on page 2-33 for more information).

Your application may call the `EnterMovies` function multiple times for a given A5 world, as long as you balance each invocation of `EnterMovies` with an invocation of `ExitMovies`.

SPECIAL CONSIDERATIONS

The Movie Toolbox identifies an application by the value in the A5 register. If you are writing a stand-alone code resource, you must ensure that A5 is the same whenever you call any Movie Toolbox functions.

ERROR CODES

Memory Manager errors

SEE ALSO

Listing 2-3 on page 2-39 provides an example of the `EnterMovies` function.

ExitMovies

QuickTime calls the `ExitMovies` function automatically when your application quits—you only need to call this function if you finish with the Movie Toolbox long before your application is ready to quit. As a general rule, your application should not use this function.

```
pascal void ExitMovies (void);
```

DESCRIPTION

When you call the `ExitMovies` function, the Movie Toolbox releases the private storage (which may be significant) that was allocated when you called the `EnterMovies` function, which is described in the previous section.

SPECIAL CONSIDERATIONS

Before calling the `ExitMovies` function, be sure that you have closed your connections to any components that use the Movie Toolbox (such as movie controllers, sequence grabbers, and so on).

ERROR CODES

None

Error Functions

The Movie Toolbox provides a number of functions that allow your application to examine result codes generated by toolbox functions. In addition, the Movie Toolbox allows your application to provide a function that performs custom error notification. This section discusses these error functions.

IMPORTANT

The Movie Toolbox introduces an additional error-reporting mechanism. In addition to returning errors as function results, the Movie Toolbox functions return error indications to calling applications by setting one of two values that are private to the Movie Toolbox: a current error value or a sticky error value. Your application can retrieve these values by calling the `GetMoviesError` or `GetMoviesStickyError` functions described in this section. To let you know whether there is an error indication, the heading “ERROR CODES” may appear with the entry “None” in function descriptions throughout this chapter. s

The Movie Toolbox maintains two error values for your application: the current error and the sticky error. The **current error** value contains the result code from the last Movie Toolbox function. The toolbox updates the current error value each time your application calls a Movie Toolbox function. Your application may call the `GetMoviesError` function to obtain the current error value after calling any Movie Toolbox function. Many Movie Toolbox functions do not return an error as a function result—you must use the `GetMoviesError` function to obtain the result code. Even if a function explicitly returns an error as a function result, that result is also available using the `GetMoviesError` function.

The Movie Toolbox saves a result code in the **sticky error** value. Your application clears the sticky error value by calling the `ClearMoviesStickyError` function. The Movie Toolbox then places the first nonzero result code from any toolbox function used by your application into the sticky error value. The Movie Toolbox does not replace the value in the sticky error value until your application clears the value again. Your application uses the `GetMoviesStickyError` function to obtain the result code stored in the sticky error value. In this manner, you can preserve and retrieve important result code information.

Your application uses the `SetMoviesErrorProc` function to designate an error function. The Movie Toolbox calls this error function each time there is an error.

GetMoviesError

The `GetMoviesError` function returns the contents of the current error value and resets the current error value to 0.

```
pascal OSErr GetMoviesError (void);
```

DESCRIPTION

The current error value contains the result code from the previous Movie Toolbox function. Most Movie Toolbox functions do not return an error as a function result—you must use the `GetMoviesError` function to obtain the result code. Even if a function explicitly returns an error as a function result, that result is also available using the `GetMoviesError` function.

ERROR CODES

Any Movie Toolbox result code (see “Summary of the Movie Toolbox” at the end of this chapter)

GetMoviesStickyError

The `GetMoviesStickyError` function returns the contents of the sticky error value. The sticky error value contains the first nonzero result code from any Movie Toolbox function that you called after having cleared the sticky error with the `ClearMoviesStickyError` function.

```
pascal OSErr GetMoviesStickyError (void);
```

DESCRIPTION

The Movie Toolbox does not clear the sticky error value when you call the `GetMoviesStickyError` function. Your application clears the sticky error value by calling the `ClearMoviesStickyError` function, which is described in the next section.

ERROR CODES

Any Movie Toolbox result code (see “Summary of the Movie Toolbox” at the end of this chapter)

ClearMoviesStickyError

The `ClearMoviesStickyError` function clears the sticky error value.

```
pascal void ClearMoviesStickyError (void);
```

DESCRIPTION

The Movie Toolbox does not place a result code into the sticky error value until the field has been cleared. Your application should clear the sticky error value to ensure that it does not contain a stale result code.

ERROR CODES

None

SetMoviesErrorProc

The Movie Toolbox allows applications to perform custom error notification. Your application must identify its custom error-notification function to the Movie Toolbox. The `SetMoviesErrorProc` function allows you to identify your application's error-notification function. Error-notification functions can be especially useful when you are debugging your program.

```
pascal void SetMoviesErrorProc (ErrorProcPtr errProc,
                                long refcon);
```

`errProcPtr`
: Points to your error-notification function, `MyErrProc`.

  The entry point to your error-notification function must take the following form:

  ```
  pascal void MyErrProc (OSErr theErr, long refCon);
  ```

  See "Application-Defined Functions" beginning on page 2-354 for details on the parameters.

`refcon`
: Contains a reference constant value. The Movie Toolbox passes this reference constant to your error-notification function each time it calls your function.

DESCRIPTION

Once you have identified an error-notification function, the Movie Toolbox calls your function each time the current error value is to be set to a nonzero value. The Movie Toolbox manages the sticky error value. The Movie Toolbox calls your error-notification function only in response to errors generated by the Movie Toolbox.

SPECIAL CONSIDERATIONS

The `SetMoviesErrorProc` function is just for debugging.

ERROR CODES

None

Movie Functions

The Movie Toolbox provides a set of functions that allow your application to create, access, and convert movie files. Movie 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.

Before your application can play a movie, you must first open the file that contains the movie. Your application can use the `OpenMovieFile` function (described on page 2-98) 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," which begins on page 2-100, 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.

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

NewMovieFromFile

The NewMovieFromFile function creates a movie in memory from a resource that is stored in a movie file. Your application specifies the movie file with the file reference number that was returned by the OpenMovieFile function, which is described on page 2-98. Your application can use the NewMovieFromHandle function, described in the next section, to load a movie from a handle. Once you have opened a movie file and loaded a movie, your application can proceed to work with the movie.

```
pascal OSErr NewMovieFromFile (Movie *theMovie, short resRefNum,
                               short *resId,
                               StringPtr resName,
                               short newMovieFlags,
                               Boolean *dataRefWasChanged);
```

theMovie Contains a pointer to a field that is to receive the new movie's identifier. If the function cannot load the movie, the returned identifier is set to nil.

resRefNum Identifies the movie file from which the movie is to be loaded. Your application obtains this value from the OpenMovieFile function, described on page 2-98.

resId Contains a pointer to a field that specifies the resource containing the movie data that is to be loaded. If the field referred to by the resId parameter is set to 0, the Movie Toolbox loads the first movie resource it finds in the specified file. The toolbox then returns the movie's resource ID number in the field referred to by the resId parameter. The following enumerated constant is available:

movieInDataForkResID
: Forces the movie to come out of the data fork. If the resource was stored in the file's data fork, the Movie Toolbox sets the returned value to movieInDataForkResID (–1). In this case, you cannot add a movie resource to the file unless you create a resource fork in the movie file.

If the resId parameter is set to nil, the Movie Toolbox loads the first movie resource it finds in the specified file and does not return that resource's ID number.

resName Points to a character string that is to receive the name of the movie resource that is loaded. If you set the resName parameter to nil, the toolbox does not return the resource name.

newMovieFlags

Controls the operation of the NewMovieFromFile function. The following flags are available (be sure to set unused flags to 0):

newMovieActive

Controls whether the new movie is active. Set this flag to 1 to make the new movie active. You can make a movie active or inactive by calling the SetMovieActive function, which is described on page 2-145.

newMovieDontResolveDataRefs

Controls how completely the Movie Toolbox resolves data references in the movie resource. If you set this flag to 0, the toolbox tries to completely resolve all data references in the resource. This may involve searching for files on multiple volumes. If you set this flag to 1, the Movie Toolbox only looks in the specified file.

If the Movie Toolbox cannot completely resolve all the data references, it still returns a valid movie identifier. In this case, the Movie Toolbox also sets the current error value to couldNotResolveDataRef.

newMovieDontAskUnresolvedDataRefs

Controls whether the Movie Toolbox asks the user to locate files. If you set this flag to 0, the Movie Toolbox asks the user to locate files that it cannot find. If the Movie Toolbox cannot locate a file even with the user's help, the function returns a valid movie identifier and sets the current error value to couldNotResolveDataRef.

newMovieDontAutoAlternate

Controls whether the Movie Toolbox automatically selects enabled tracks from alternate track groups. If you set this flag to 1, the Movie Toolbox does not automatically select tracks for the movie—you must enable tracks yourself.

dataRefWasChanged

Contains a pointer to a Boolean value. The Movie Toolbox sets the Boolean to indicate whether it had to change any data references while resolving them. The toolbox sets the Boolean value to true if any references were changed. Use the UpdateMovieResource function (described on page 2-103) to preserve these changes.

Set the dataRefWasChanged parameter to nil if you do not want to receive this information. See "Creating Tracks and Media Structures" beginning on page 2-150 for more information about data references.

DESCRIPTION

The Movie Toolbox sets many movie characteristics to default values. If you want to change these defaults, your application must call other Movie Toolbox functions. For example, the Movie Toolbox sets the movie's graphics world to the one that is active when you call `NewMovieFromFile`. To change the graphics world for the new movie, your application should use the `SetMovieGWorld` function, which is described on page 2-159.

SPECIAL CONSIDERATIONS

The Movie Toolbox automatically sets the movie's graphics world based upon the current graphics port. Be sure that your application's graphics world is valid before you call this function.

ERROR CODES

`badImageDescription`	–2001	Problem with an image description
`badPublicMovieAtom`	–2002	Movie file corrupted
`cantFindHandler`	–2003	Cannot locate a handler
`cantOpenHandler`	–2004	Cannot open a handler

File Manager errors
Memory Manager errors
Resource Manager errors

NewMovieFromHandle

The `NewMovieFromHandle` function creates a movie in memory from a movie resource or a handle you obtained from the `PutMovieIntoHandle` function.

```
pascal OSErr NewMovieFromHandle (Movie *theMovie, Handle h,
                                 short newMovieFlags,
                                 Boolean *dataRefWasChanged);
```

`theMovie` Contains a pointer to a field that is to receive the new movie's identifier. If the function cannot load the movie, the returned identifier is set to `nil`.

`h` Contains a handle to the movie resource from which the movie is to be loaded.

`newMovieFlags`
Controls the operation of the `NewMovieFromHandle` function. The following flags are available (be sure to set unused flags to 0):

`newMovieActive`
Controls whether the new movie is active. Set this flag to 1 to make the new movie active. You can make a movie active or inactive by calling the `SetMovieActive` function, which is described on page 2-145.

newMovieDontResolveDataRefs
: Controls how completely the Movie Toolbox resolves data references in the movie resource. If you set this flag to 0, the toolbox tries to completely resolve all data references in the resource. This may involve searching for files on remote volumes. If you set this flag to 1, the Movie Toolbox only looks in the specified file.

  If the Movie Toolbox cannot completely resolve all the data references, it still returns a valid movie identifier. In this case, the Movie Toolbox also sets the current error value to `couldNotResolveDataRef`.

newMovieDontAskUnresolvedDataRefs
: Controls whether the Movie Toolbox asks the user to locate files. If you set this flag to 0, the Movie Toolbox asks the user to locate files that it cannot find on available volumes. If the Movie Toolbox cannot locate a file even with the user's help, the function returns a valid movie identifier and sets the current error value to `couldNotResolveDataRef`.

newMovieDontAutoAlternate
: Controls whether the Movie Toolbox automatically selects enabled tracks from alternate track groups. If you set this flag to 1, the Movie Toolbox does not automatically select tracks for the movie—you must enable tracks yourself.

dataRefWasChanged
: Contains a pointer to a Boolean value. The Movie Toolbox sets the Boolean value to indicate whether it had to change any data references in order to resolve them. The toolbox sets the Boolean value to `true` if any references were changed. Set the `dataRefWasChanged` parameter to `nil` if you do not want to receive this information.

DESCRIPTION

The `NewMovieFromHandle` function returns the new movie's identifier. If the function cannot create the movie, the function sets the returned identifier to `nil`.

Your application can use the `NewMovieFromFile` function, described in the previous section, to load a movie from a movie file that was opened with the `OpenMovieFile` function. If you are loading a movie from a resource, use the `NewMovieFromFile` function instead. The Movie Toolbox uses information about the resource file when it resolves data references in the movie.

The Movie Toolbox sets many movie characteristics to default values. If you want to change these defaults, your application must call other Movie Toolbox functions. For example, the Movie Toolbox sets the movie's graphics world to the one that is active when you call `NewMovieFromHandle`. To change the graphics world for the new movie, your application should use the `SetMovieGWorld` function, which is described on page 2-159.

SPECIAL CONSIDERATIONS

The Movie Toolbox automatically sets the movie's graphics world based upon the current graphics port. Be sure that your application's graphics world is valid before you call this function.

ERROR CODES

`badImageDescription`	–2001	Problem with an image description
`badPublicMovieAtom`	–2002	Movie file corrupted
`cantFindHandler`	–2003	Cannot locate a handler
`cantOpenHandler`	–2004	Cannot open a handler

File Manager errors
Memory Manager errors
Resource Manager errors

NewMovie

The `NewMovie` function creates a new movie in memory. The Movie Toolbox initializes the data structures for the new movie, which contains no tracks. Your application assigns the data to the movie by calling the functions that are described later in "Creating Tracks and Media Structures" beginning on page 2-150.

```
pascal Movie NewMovie (long newMovieFlags);
```

`newMovieFlags`
Specifies control information for the new movie. The following flags are available (be sure to set unused flags to 0):

`newMovieActive`
Controls whether the new movie is active. Set this flag to 1 to make the new movie active. A movie that does not have any tracks can still be active. When the Movie Toolbox tries to play the movie, no images are displayed, because there is no movie data. You can make a movie active or inactive by calling the `SetMovieActive` function, which is described on page 2-145.

`newMovieDontAutoAlternate`
Controls whether the Movie Toolbox automatically selects enabled tracks from alternate track groups. If you set this flag to 1, the Movie Toolbox does not automatically select tracks for the movie—you must enable tracks yourself.

DESCRIPTION

The `NewMovie` function returns the identifier for the new movie. If the function fails, the returned identifier is set to `nil`. Use the `GetMoviesError` function (described on page 2-85) to obtain the result code.

The Movie Toolbox sets many movie characteristics to default values. If you want to change these defaults, your application must call other Movie Toolbox functions. For example, the Movie Toolbox sets the movie's graphics world to the one that is active when you call `NewMovie`. To change the graphics world for the new movie, your application should use the `SetMovieGWorld` function, which is described on page 2-159.

The default QuickTime movie time scale is 600 units per second; however, this number may change in the future. The default time scale was chosen because it is convenient for working with common video frame rates of 30, 25, 24, 15, 12, 10, and 8.

You should use the `NewMovie` function only if you have not created a new movie and movie file by calling the `CreateMovieFile` function.

s **WARNING**
The Movie Toolbox automatically sets the movie's graphics world based upon the current graphics port. Be sure that your application's graphics port is valid before you call this function. s

ERROR CODES

`movieToolboxUninitialized`	–2020	You haven't initialized the Movie Toolbox

Memory Manager errors

ConvertFileToMovieFile

The `ConvertFileToMovieFile` takes a specified file and converts it to a movie file.

```
pascal OSErr ConvertFileToMovieFile (const FSSpec *inputFile,
                                     const FSSpec *outputFile,
                                     OSType creator,
                                     ScriptCode scriptTag,
                                     short *resID, long flags,
                                     ComponentInstance userComp,
                                     MovieProgressProcPtr proc,
                                     long refCon);
```

`inputFile` Contains a pointer to the file system specification for the file to be converted into a movie file.

`outputFile` Contains a pointer to the file specification for the destination movie file.

`creator` Specifies the creator value for the file if it is a new one.

`scriptTag` Specifies the script in which the movie file should be converted. Use the Script Manager constant `smSystemScript` to use the system script; use the `smCurrentScript` constant to use the current script. See *Inside Macintosh: Text* for more information about scripts and script tags.

`resID` Contains a pointer to a field that is to receive the resource ID of the file to be converted. If you don't want to receive the resource ID, set this parameter to `nil`.

`flags` Controls movie file conversion flags. The following value is valid:

`createMovieFileDeleteCurFile`
Indicates whether to delete an existing file. If you set this flag to 1, the Movie Toolbox deletes the file (if it exists) before converting the new movie file. If you set this flag to 0 and the file specified by the `fileSpec` parameter already exists, the Movie Toolbox uses the existing file. In this case, the toolbox ensures that the file has both a data and a resource fork.

`userComp` Indicates a component or component instance of the movie export component you want to perform the conversion. Otherwise, set this parameter to 0 for the Movie Toolbox to choose the appropriate component. If you pass in a component instance, it will be used by `ConvertFileToMovieFile`. This allows you to communicate directly with the component before using this function to establish any conversion parameters. If you pass in a component ID, an instance is created and closed within this function. For details on movie export components, see *Inside Macintosh: QuickTime Components*.

`proc` Points to your progress function. To remove a movie's progress function, set this parameter to `nil`. Set this parameter to –1 for the Movie Toolbox to provide a default progress function. See "Progress Functions," which begins on page 2-354, for the interface your progress function must support.

`refCon` Specifies a reference constant. The Movie Toolbox passes this value to your progress function.

DESCRIPTION

Because some conversions may take a nontrivial amount of time, you can pass a standard movie progress function in the `proc` and `refCon` parameters.

ConvertMovieToFile

The `ConvertMovieToFile` function takes a specified movie (or a single track within that movie) and converts it into a specified file and type.

```
pascal OSErr ConvertMovieToFile(Movie theMovie, Track onlyTrack,
                                const FSSpec *outputFile,
                                OSType fileType, OSType creator,
                                ScriptCode scriptTag,
                                short *resID, long flags,
                                ComponentInstance userComp);
```

`theMovie` Specifies the source movie for this conversion operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

`onlyTrack` Specifies the track within the source movie for this conversion operation. To specify all tracks, set the value of this parameter to 0.

`outputFile` Contains a pointer to the file specification for the destination file.

`fileType` Specifies the data type of the destination file for the movie specified in the parameter `theMovie`.

`creator` Specifies the creator value for the output file if it is a new one.

`scriptTag` Specifies the script into which the movie should be converted if the output file is a new one. Use the Script Manager constant `smSystemScript` to use the system script; use the `smCurrentScript` constant to use the current script. See *Inside Macintosh: Text* for more information about scripts and script tags.

`resID` Contains a pointer to a field that is to receive the resource ID of the open movie. If you don't want to receive this information, set the `resID` parameter to `nil`.

`flags` Set this parameter to 0.

`userComp` If you want a particular movie export component to perform the conversion, you may pass the component or an instance of that component in this parameter. Otherwise, set it to 0 to allow the Movie Toolbox to use the appropriate component. If you pass in a component instance, it is used by `ConvertMovieToFile`. This allows you to communicate directly with the component before making this call to establish any conversion parameters. If you pass in a component ID, an instance is created and closed within this call.

DisposeMovie

The `DisposeMovie` function frees any memory being used by a movie, including the memory used by the movie's tracks and media structures. Your application should call this function when it is done working with a movie.

```
pascal void DisposeMovie (Movie theMovie);
```

`theMovie` Identifies the movie to be freed. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, or `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

SPECIAL CONSIDERATIONS

Do not dispose of a movie if it has any special clients—for example, if it has an attached movie controller component. Only dispose of the movie after any clients are done with it.

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid

CreateMovieFile

The `CreateMovieFile` function creates an open movie file, opens the movie file, creates an empty movie which references the file, and opens the movie file with write permission.

```
pascal OSErr CreateMovieFile (const FSSpec *fileSpec,
                              OSType creator,
                              ScriptCode scriptTag,
                              long createMovieFileFlags,
                              short *resRefNum,
                              Movie *newMovie);
```

`fileSpec` Contains a pointer to the file system specification for the movie file to be created.

`creator` Specifies the creator value for the new file.

`scriptTag` Specifies the script in which the movie file should be created. Use the Script Manager constant `smSystemScript` to use the system script; use the `smCurrentScript` constant to use the current script. See *Inside Macintosh: Text* for more information about scripts and script tags.

createMovieFileFlags
: Controls movie-file creation flags. The following flags are available:

  createMovieFileDeleteCurFile
  : Indicates whether to delete an existing file. If you set this flag to 1, the Movie Toolbox deletes the file (if it exists) before creating the new movie file. If you set this flag to 0 and the file specified by the fileSpec parameter already exists, the Movie Toolbox uses the existing file. In this case, the toolbox ensures that the file has both a data and a resource fork.

  createMovieFileDontCreateMovie
  : Controls whether the CreateMovieFile function creates a new movie in the movie file. If you set this flag to 1, the Movie Toolbox does not create a movie in the new movie file. In this case, the function ignores the newMovie parameter. If you set this flag to 0, the Movie Toolbox creates a movie and returns the movie identifier in the field referred to by the newMovie parameter.

  createMovieFileDontOpenFile
  : Controls whether the CreateMovieFile function opens the new movie file. If you set this flag to 1, the Movie Toolbox does not open the new movie file. In this case, the function ignores the resRefNum parameter. If you set this flag to 0, the Movie Toolbox opens the new movie file and returns its reference number into the field referred to by the resRefNum parameter.

  newMovieActive
  : Controls whether the new movie is active. Set this flag to 1 to make the new movie active. A movie that does not have any tracks can still be active. When the Movie Toolbox tries to play the movie, no images are displayed, because there is no movie data. You can make a movie active or inactive by calling the SetMovieActive function, which is described on page 2-145.

  newMovieDontAutoAlternate
  : Controls whether the Movie Toolbox automatically selects enabled tracks from alternate track groups. If you set this flag to 1, the Movie Toolbox does not automatically select tracks for the movie—you must enable tracks yourself.

resRefNum
: Contains a pointer to a field that is to receive the file reference number for the opened movie file. Your application must use this value when calling other Movie Toolbox functions that work with movie files. If you set this parameter to nil, the Movie Toolbox creates the movie file but does not open the file.

`newMovie` Contains a pointer to a field that is to receive the identifier of the new movie. The `CreateMovieFile` function returns the identifier of the new movie. If the function could not create a new movie, it sets this returned value to `nil`. If you set this parameter to `nil`, the Movie Toolbox does not create a movie.

ERROR CODES

`movieToolboxUninitialized`	–2020	You haven't initialized the Movie Toolbox

File Manager errors
Memory Manager errors

SEE ALSO

You can delete a movie file by calling the `DeleteMovieFile` function, which is described on page 2-100.

Your application can use the functions described in "Creating Tracks and Media Structures," which begins on page 2-150, to place movie data into the new movie file.

OpenMovieFile

The `OpenMovieFile` function opens a specified movie file. Your application identifies the movie file with a file system specification.

```
pascal OSErr OpenMovieFile (const FSSpec *fileSpec,
                            short *resRefNum, char perms);
```

`fileSpec` Contains a pointer to the file system specification for the movie file to be opened.

`resRefNum` Contains a pointer to a field that is to receive the file reference number for the opened movie file. Your application must use this value when calling other Movie Toolbox functions that work with movie files. This reference number refers to the file fork that contains the movie resource—if the movie is stored in the data fork of the file, the returned reference number corresponds to the data fork.

`perms` Specifies the permission level for the file. If your application is only going to play the movie that is stored in the file, you can open the file with read permission. If you plan to add data to the file or change data in the file, you should open the file with write permission. Supply a valid File Manager permission value. See *Inside Macintosh: Files* for valid values.

DESCRIPTION

Your application must open a movie file before reading movie data from it or writing movie data to it. You can open a movie file more than once—be sure to call `CloseMovieFile` (described in the next section) once for each time you call `OpenMovieFile`.

Note that opening the movie file with write permission does not prevent other applications from reading data from the movie file.

If the specified file has a resource fork, the `OpenMovieFile` function opens the resource fork and returns a file reference number to the resource fork. If the movie file does not have a resource fork (that is, it is a single-fork movie file—see the chapter "Movie Resource Formats" in this book for more information), the `OpenMovieFile` function opens the data fork instead. In this case, your application cannot use the `AddMovieResource` function (described on page 2-102) with the movie file.

ERROR CODES

`movieToolboxUninitialized`	–2020	You haven't initialized the Movie Toolbox

File Manager errors
Memory Manager errors

CloseMovieFile

The `CloseMovieFile` function closes an open movie file.

```
pascal OSErr CloseMovieFile (short resRefNum);
```

`resRefNum` Specifies the movie file to close. Your application obtains this reference number from the `OpenMovieFile` function, which is described in the previous section.

DESCRIPTION

Your application should call this function when you are done working with a movie file. You must call this function once each time you open a movie file. You can still use the movie. If you are not editing the movie, it is advisable to close it.

ERROR CODES

File Manager errors

DeleteMovieFile

The `DeleteMovieFile` function deletes a movie file.

```
pascal OSErr DeleteMovieFile (const FSSpec *fileSpec);
```

`fileSpec` Contains a pointer to the file system specification for the movie file to be deleted.

DESCRIPTION

Do not use the file system to delete movie files. The Movie Toolbox maintains references between files.

ERROR CODES

File Manager errors

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 (described on page 2-88 and page 2-109, respectively). 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 described in "Functions for Editing Movies" beginning on page 2-242. 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` function. If you want to create a single-fork movie file, use the `FlattenMovieData` function.

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.

HasMovieChanged

The `HasMovieChanged` function allows your application to determine whether a movie has changed and needs to be saved.

```
pascal Boolean HasMovieChanged (Movie theMovie);
```

`theMovie` Specifies the movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

DESCRIPTION

The `HasMovieChanged` function returns a Boolean value that reflects the contents of the movie changed flag. The function sets the returned value to `true` if the movie has been changed in such a way that it should be saved. Otherwise, the returned value is set to `false`.

Your application can clear the movie changed flag, indicating that the movie has not changed, by calling the `ClearMovieChanged` function, which is described in the next section.

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid

SEE ALSO

Both the `AddMovieResource` function (described on page 2-102) and the `UpdateMovieResource` function (described on page 2-103) update the movie file and clear the movie changed flag, indicating that the movie has not been changed.

ClearMovieChanged

The `ClearMovieChanged` function sets the movie changed flag to indicate that the movie has not been changed.

```
pascal void ClearMovieChanged (Movie theMovie);
```

`theMovie` Specifies the movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid

SEE ALSO

Your application can read the contents of the movie changed flag by calling the `HasMovieChanged` function, which is described in the previous section. Both the `AddMovieResource` and `UpdateMovieResource` functions also clear the movie changed flag.

AddMovieResource

The `AddMovieResource` function adds a movie resource to a specified resource file. Your application identifies the movie to be added to the movie file.

```
pascal OSErr AddMovieResource (Movie theMovie, short resRefNum,
                               short *resId,
                               const StringPtr resName);
```

`theMovie` Specifies the movie you wish to add to the movie file. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

`resRefNum` Identifies the movie file to which the resource is to be added. Your application obtains this value from the `OpenMovieFile` function, described on page 2-98. The movie file specified by this parameter cannot be a single-fork movie file.

`resId` Contains a pointer to a field that contains the resource ID number for the new resource. If the field referred to by the `resId` parameter is set to 0, the Movie Toolbox assigns a unique resource ID number to the new resource. The toolbox then returns the movie's resource ID number in the field referred to by the `resId` parameter. The `AddMovieResource`

function assigns resource ID numbers sequentially, starting at 128. If the `resId` parameter is set to `nil`, the Movie Toolbox assigns a unique resource ID number to the new resource and does not return that resource's ID value.

`resName` Points to a character string that contains the name of the movie resource. If you set the `resName` parameter to `nil`, the toolbox creates an unnamed resource.

DESCRIPTION

The `AddMovieResource` function adds the movie to the file, effectively saving any changes you have made to the movie. This function does not work with single-fork movie files.

After updating the movie file, `AddMovieResource` clears the movie changed flag, indicating that the movie has not been changed.

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid

File Manager errors
Memory Manager errors
Resource Manager errors

UpdateMovieResource

The `UpdateMovieResource` function replaces the contents of a movie resource in a specified movie file. You specify the movie that is to be placed into the resource.

This function can accommodate single-fork movie files.

```
pascal OSErr UpdateMovieResource (Movie theMovie, short resRefNum,
                                  short resId,
                                  const StringPtr resName);
```

`theMovie` Specifies the movie you wish to place in the movie file. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

`resRefNum` Identifies the movie file that contains the resource to be changed. Your application obtains this value from the `OpenMovieFile` function, described on page 2-98. If this parameter specifies a single-fork movie file using the `movieInDataForResID(–1)` constant, the Movie Toolbox places the movie resource into the file's data fork.

`resId` Specifies the resource to be changed.

resName Points to a new name for the resource. If you do not want to change the resource's name, set this parameter to nil.

DESCRIPTION

After updating the movie file, the UpdateMovieResource function clears the movie changed flag, indicating that the movie has not been changed.

ERROR CODES

invalidMovie	–2010	This movie is corrupted or invalid

File Manager errors
Memory Manager errors
Resource Manager errors

RemoveMovieResource

The RemoveMovieResource function removes a movie resource from a specified movie file.

```
pascal OSErr RemoveMovieResource (short resRefNum, short resId);
```

resRefNum Identifies the movie file that contains the movie resource. Your application obtains this value from the OpenMovieFile function, described on page 2-98.

resId Specifies the resource to be removed.

ERROR CODES

File Manager errors
Resource Manager errors

PutMovieIntoHandle

The PutMovieIntoHandle function creates a new movie resource for you. You can use this handle to store a QuickTime movie in a specialized storage format.

```
pascal OSErr PutMovieIntoHandle (Movie theMovie,
                                 Handle publicMovie);
```

`theMovie` Specifies the movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

`publicMovie`
Contains the handle that is to receive the new movie resource. The `PutMovieIntoHandle` function places the new movie resource into this handle. The function resizes the handle if necessary.

DESCRIPTION

Note that you cannot use this new movie with other Movie Toolbox functions, except for the `NewMovieFromHandle` function. You can use the `NewMovieFromHandle` function, described on page 2-90, to load a movie from a handle.

SPECIAL CONSIDERATIONS

Movies saved using `PutMovieIntoHandle` contain less robust data references than those created using the `AddMovieResource` or `PutMovieIntoDataFork` functions (described on page 2-102 and page 2-110, respectively).

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid

Memory Manager errors

FlattenMovie

The `FlattenMovie` function creates a new movie file containing a specified movie. This file also contains all the data for the movie—that is, the Movie Toolbox resolves any data references and includes the corresponding movie data in the new movie file.

```
pascal void FlattenMovie (Movie theMovie, long movieFlattenFlags,
                          const FSSpec *theFile,
                          OSType creator, ScriptCode scriptTag,
                          long createMovieFileFlags,
                          short *resId, const StringPtr resName);
```

`theMovie` Specifies the movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

`movieFlattenFlags`
Controls the process of adding movie data to the new movie file. The following flags are available (be sure to set unused flags to 0):

`flattenAddMovieToDataFork`
Causes the movie to be placed in the data fork of the new movie file, as well as in the resource fork. You may use this flag to create movie files that are more easily moved to other computer systems from your Macintosh.

`flattenDontInterleaveFlatten`
Allows you to disable the Movie Toolbox's data storage optimizations. By default, the Movie Toolbox stores movie data in a format that is optimized for playback. Set this flag to 1 to disable these optimizations.

`flattenActiveTracksOnly`
Causes the Movie Toolbox to add only enabled movie tracks to the new movie file. You can use the `SetTrackEnabled` function, described on page 2-147, to enable and disable movie tracks.

`theFile` Contains a pointer to the file system specification for the movie file to be created.

`creator` Specifies the creator value for the new file.

`scriptTag` Specifies the script in which the movie file should be created. Set this parameter to the Script Manager constant `smSystemScript` to use the system script; set it to `smCurrentScript` to use the current script. See *Inside Macintosh: Text* for more information about scripts and script tags.

`createMovieFileFlags`
Controls file creation options. The following flag is available:

`createMovieFileDeleteCurFile`
Indicates whether to delete an existing file. If you set this flag to 1, the Movie Toolbox deletes the file (if it exists) before creating the new movie file. If this flag is set to 0 and the file specified by the `fileSpec` parameter already exists, the Movie Toolbox uses the existing file. In this case, the toolbox ensures that the file has both a data and a resource fork. If this flag is not set, the data is appended to the file.

`resId` Contains a pointer to a field that contains the resource ID number for the new resource. If the field referred to by the `resId` parameter is set to 0, the Movie Toolbox assigns a unique resource ID number to the new resource. The toolbox then returns the movie's resource ID number in the field referred to by the `resId` parameter. The Movie Toolbox assigns resource ID numbers sequentially, starting at 128. If the `resId` parameter is set to `nil`, the Movie Toolbox assigns a unique resource ID number to the new resource and does not return that resource's ID value.

`resName` Points to a character string with the name of the movie resource. If you set the `resName` parameter to `nil`, the toolbox creates an unnamed resource.

DESCRIPTION

The toolbox places the movie resource into the resource fork of the movie file. The Movie Toolbox does not alter the source movie.

The Movie Toolbox calls your progress function during long operations.

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid
`progressProcAborted`	–2019	Your progress function returned an error
`cantCreateSingleForkFile`	–2022	Error trying to create a single-fork file

File Manager errors
Memory Manager errors
Resource Manager errors

FlattenMovieData

The `FlattenMovieData` function creates a new movie file and creates a new movie that contains all of its movie data. However, unlike the `FlattenMovie` function described in the previous section, this function does not add the new movie resource to the new movie file. Instead, the `FlattenMovieData` function returns the new movie to your application. Your application must dispose of the returned movie.

```
pascal Movie FlattenMovieData (Movie theMovie,
                               long movieFlattenFlags,
                               const FSSpec *theFile,
                          OSType creator,
                          ScriptCode scriptTag,
                          long createMovieFileFlags);
```

`theMovie` Specifies the movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

`movieFlattenFlags`
Controls the process of adding movie data to the new movie file. These flags affect how the toolbox adds movies to the new movie file later. The following flags are available (be sure to set unused flags to 0):

`flattenAddMovieToDataFork`
Causes the movie to be placed in the data fork of the new movie file. You may use this flag to create single-fork movie files, which can be more easily moved to other computer systems from your Macintosh.

flattenDontInterleaveFlatten
: Allows you to disable the Movie Toolbox's data storage optimizations. By default, the Movie Toolbox stores movie data in a format that is optimized for the storage device. Set this flag to 1 to disable these optimizations.

flattenActiveTracksOnly
: Causes the Movie Toolbox to add only enabled movie tracks to the new movie file. You can use the SetTrackEnabled function, which is described on page 2-147, to enable and disable movie tracks.

theFile
: Contains a pointer to the file system specification for the movie file to be created.

creator
: Specifies the creator value for the new file.

scriptTag
: Specifies the script in which the movie file should be created. Set this parameter to smSystemScript to use the system script; set it to smCurrentScript to use the current script. See *Inside Macintosh: Text* for more information about scripts and script tags.

creationFlags
: Controls file creation options. The following flag is available:

  createMovieFileDeleteCurFile
  : Indicates whether to delete an existing file. If you set this flag to 1, the Movie Toolbox deletes the file (if it exists) before creating the new movie file. If this flag is set to 0 and the file specified by the fileSpec parameter already exists, the Movie Toolbox uses the existing file. In this case, the toolbox ensures that the file has both a data and a resource fork. If this flag isn't set, the data is appended to the file.

DESCRIPTION

The FlattenMovieData function returns the movie identifier of the new movie. If the function could not create the movie, it sets this returned identifier to nil.

You can also use this function to create a single-fork movie file. Set the flattenAddMovieToDataFork flag in the movieFlattenFlags parameter to 1. The Movie Toolbox then places the movie into the data fork of the movie file.

The Movie Toolbox calls your progress function during long operations.

The Movie Toolbox does not alter the source movie.

ERROR CODES

invalidMovie	–2010	This movie is corrupted or invalid
progressProcAborted	–2019	Your progress function returned an error
cantCreateSingleForkFile	–2022	Error trying to create a single-fork file

File Manager errors

Memory Manager errors

NewMovieFromDataFork

The NewMovieFromDataFork function enables you to retrieve a movie that is stored anywhere in the data fork of a specified file.

```
pascal OSErr NewMovieFromDataFork (Movie *theMovie,
                                   short fRefNum,
                                   long fileOffset,
                                   short newMovieFlags,
                                   Boolean *dataRefWasChanged);
```

theMovie Contains a pointer to the movie identifier for the movie to be retrieved. Your application obtains this movie identifier from such functions as NewMovie, NewMovieFromFile, and NewMovieFromHandle (described on page 2-92, page 2-88, and page 2-90, respectively).

fRefNum Contains a file reference number to a file that is already open.

fileOffset

Specifies the starting file offset of the atom in the data fork of the file specified by the fRefNum parameter.

newMovieFlags

Contains the standard flags in the newMovie enumeration.

newMovieActive

Controls whether the new movie is active. Set this flag to 1 to make the new movie active. A movie that does not have any tracks can still be active. When the Movie Toolbox tries to play the movie, no images are displayed, because there is no movie data. You can make a movie active or inactive by calling the SetMovieActive function, which is described on page 2-145.

newMovieDontAutoAlternate

Controls whether the Movie Toolbox automatically selects enabled tracks from alternate track groups. If you set this flag to 1, the Movie Toolbox does not automatically select tracks for the movie—you must enable tracks yourself.

newMovieDontResolveDataRefs

Controls how completely the Movie Toolbox resolves data references in the movie resource. If you set this flag to 0, the toolbox tries to completely resolve all data references in the resource. This may involve searching for files on remote volumes. If you set this flag to 1, the Movie Toolbox only looks in the specified file.

If the Movie Toolbox cannot completely resolve all the data references, it still returns a valid movie identifier. In this case, the Movie Toolbox also sets the current error value to couldNotResolveDataRef.

newMovieDontAskUnresolvedDataRefs
: Controls whether the Movie Toolbox asks the user to locate files. If you set this flag to 0, the Movie Toolbox asks the user to locate files that it cannot find on available volumes. If the Movie Toolbox cannot locate a file even with the user's help, the function returns a valid movie identifier and sets the current error value to couldNotResolveDataRef.

dataRefWasChanged
: Contains a pointer to a Boolean value. The Movie Toolbox sets the Boolean to indicate whether it had to change any data references while resolving them. The toolbox sets the Boolean value to true if any references were changed. Use the UpdateMovieResource function (described on page 2-103) to preserve these changes.

  Set the dataRefWasChanged parameter to nil if you do not want to receive this information. See the "Creating Tracks and Media Structures" beginning on page 2-150 for more information about data references.

ERROR CODES

badImageDescription	–2001	Problem with an image description
badPublicMovieAtom	–2002	Movie file corrupted
cantFindHandler	–2003	Cannot locate a handler
cantOpenHandler	–2004	Cannot open a handler

File Manager errors

Memory Manager errors

PutMovieIntoDataFork

The PutMovieIntoDataFork function allows you to store a movie in the data fork of a given file.

```
pascal OSErr PutMovieIntoDataFork (Movie theMovie, short fRefNum,
                                   long offset, long maxSize);
```

theMovie
: Identifies the movie to be stored in the data fork of an atom. Your application obtains this movie identifier from such functions as NewMovie, NewMovieFromFile, and NewMovieFromHandle (described on page 2-92, page 2-88, and page 2-90, respectively).

fRefNum
: Contains a file reference number for the data fork of the given file. You pass in an open write path in the fRefNum parameter.

offset
: Indicates where the movie should be written.

maxSize
: Indicates the largest number of bytes that may be written.

DESCRIPTION

If necessary, the file will be extended. If there is insufficient space to write the movie, either due to a lack of disk space or because of the limit specified in the `maxSize` parameter, this function returns a `dskFullErr` error code. If there is no limit on how much space the movie may take up in the file, pass 0 in the `maxSize` parameter.

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid

Memory Manager errors
File Manager errors

Controlling Movie Playback

This section describes a number of high-level functions provided by the Movie Toolbox that allow your application to play movies. For information about how to control a movie's playback rate, see "Working with Movie Time" beginning on page 2-184.

You can use the `StartMovie` and `StopMovie` functions to start and stop movies.

The Movie Toolbox provides functions that can be used to control your position within a movie. You can use two functions, `GoToBeginningOfMovie` and `GoToEndOfMovie`, to set the position at either the beginning or the end of a movie. These functions are described in this section. Functions that work with time bases, such as `SetMovieTimeValue` and `GetMovieTimeScale`, can be used to control the current position anywhere within a movie. These advanced functions are described in "Functions That Modify Movie Properties" beginning on page 2-157.

StartMovie

The `StartMovie` function starts the movie playing from the current movie time, which is where the movie last stopped playing. Before playing the movie, the Movie Toolbox makes the movie active, prerolls the movie, and sets the movie to its preferred playback rate. You can use the `SetMoviePreferredRate` function (described on page 2-130) to change this setting.

```
pascal void StartMovie (Movie theMovie);
```

`theMovie` Specifies the movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

DESCRIPTION

Note that a movie's current time is saved when a movie is stored in a movie file. Therefore, your application should appropriately position a movie before playing the movie—use the `GoToBeginningOfMovie` function (described on page 2-113) to set a movie to play from its start.

You are not required to call `StartMovie` to start a movie. This function is included merely for convenience.

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid

Memory Manager errors

SEE ALSO

You can also start a movie playing by calling the `SetMovieRate` function (described on page 2-187) and setting the movie's rate to a nonzero value.

StopMovie

The `StopMovie` function stops the playback of a movie.

```
pascal void StopMovie (Movie theMovie);
```

`theMovie` Specifies the movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid

SEE ALSO

You can use the `StartMovie` function described in the previous section to resume playing.

GoToBeginningOfMovie

The `GoToBeginningOfMovie` function repositions a movie to play from its start.

```
pascal void GoToBeginningOfMovie (Movie theMovie);
```

`theMovie` Specifies the movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

DESCRIPTION

If you have defined an active movie segment, the `GoToBeginningOfMovie` function repositions to the start of the active segment. The **active movie segment** is the part of the movie that your application is interested in playing. By default, the active movie segment is set to be the entire movie. You may wish to change this to be some segment of the movie—for example, if you wish to play a user's selection repeatedly. By setting the active movie segment, you guarantee that the Movie Toolbox uses no samples from outside of that range while playing the movie.

If the movie is in preview mode, the function goes to the start of the preview segment of the movie. In all other cases, this function moves you to the start of the movie, where the movie time value is 0.

SPECIAL CONSIDERATIONS

Movies need not be at the start position when they are saved. The Movie Toolbox stores a movie's time position in the movie when it is saved. If you want to play a movie from the beginning, your application should call the `GoToBeginningOfMovie` function before playing a movie you have loaded from a movie file.

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid

SEE ALSO

You can use the `SetMovieActiveSegment` and `GetMovieActiveSegment` functions to work with the active segment. For details, see "Enhancing Movie Playback Performance" beginning on page 2-134.

GoToEndOfMovie

The `GoToEndOfMovie` function repositions a movie to play from its end.

```
pascal void GoToEndOfMovie (Movie theMovie);
```

`theMovie` Specifies the movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

DESCRIPTION

If you have defined an active movie segment, the `GoToEndOfMovie` function repositions the movie to the end of the active segment. If the movie is in preview mode, the function goes to the end of the preview segment of the movie. In all other cases, this function moves you to the end of the movie.

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid

SEE ALSO

You can use the `SetMovieActiveSegment` and `GetMovieActiveSegment` functions to work with the active segment. For details, see "Enhancing Movie Playback Performance" beginning on page 2-134.

Movie Posters and Movie Previews

A QuickTime movie may contain a preview and a poster. A movie preview is a very short version of a movie, typically less than five seconds in duration. The preview is intended to give the user an idea of a movie's contents.

A movie poster is a still frame representing the movie.

This section describes the Movie Toolbox functions that allow your application to work with movie previews and movie posters.

Use the `PlayMoviePreview` function to display a movie's preview. The `PlayMoviePreview` function sets the movie into preview mode, plays the movie preview, sets the movie back to normal playback mode, and returns to your application.

Alternatively, your application can control the playback of a movie's preview. Use the `SetMoviePreviewMode` function to place a movie into preview mode. You can then use the `StartMovie` and `StopMovie` functions to control movie playback—these functions are described on page 2-111 and page 2-112, respectively. Your application can find out if a movie is in preview mode by calling the `GetMoviePreviewMode` function.

Your application can specify the starting time and duration of the movie preview with the `SetMoviePreviewTime` and `GetMoviePreviewTime` functions.

Use the `ShowMoviePoster` function to display a movie's poster. You can work with the poster's boundary rectangle using the `SetPosterBox` and `GetPosterBox` functions. Your application can work with the starting time of the poster with the `SetMoviePosterTime` and `GetMoviePosterTime` functions. Posters always have no duration.

Tracks may be specified for use in the movie, its preview, its poster, or any combination of the three. So, for example, when the Movie Toolbox plays the movie preview it uses only those tracks that are assigned to the preview. Your application controls the use of a movie's tracks with the `SetTrackUsage` function. You can find out how a track is used by calling the `GetTrackUsage` function.

SetTrackUsage

The `SetTrackUsage` function allows your application to specify whether a track is used in a movie, its preview, its poster, or a combination of these.

```
pascal void SetTrackUsage (Track theTrack, long usage);
```

`theTrack` Specifies the track for this operation. Your application obtains this track identifier from such Movie Toolbox functions as `NewMovieTrack` and `GetMovieTrack` (described on page 2-151 and page 2-204, respectively).

`usage` Contains flags that specify how the track is to be used. The following flags are defined (be sure to set unused flags to 0):

`trackUsageInMovie`
The track is used in the movie. If this flag is set to 1, the track is used in the movie.

`trackUsageInPreview`
The track is used in the preview. If this flag is set to 1, the track is used in the preview.

`trackUsageInPoster`
The track is used in the poster. If this flag is set to 1, the track is used in the poster.

ERROR CODES

`invalidTrack`	–2009	This track is corrupted or invalid

SEE ALSO

Your application can determine how a track is used by calling the `GetTrackUsage` function, which is described in the next section.

GetTrackUsage

The `GetTrackUsage` function allows your application to determine whether a track is used in a movie, its preview, its poster, or a combination of these. Your application can specify how a track is used by calling the `SetTrackUsage` function, which is described in the previous section.

```
pascal long GetTrackUsage (Track theTrack);
```

`theTrack` Specifies the track for this operation. Your application obtains this track identifier from such Movie Toolbox functions as `NewMovieTrack` and `GetMovieTrack` (described on page 2-151 and page 2-204, respectively).

DESCRIPTION

The `GetTrackUsage` function returns a long integer that contains flags indicating the track's usage. The following flags are defined (unused flags are set to 0):

`trackUsageInMovie`
The track is used in the movie. If this flag is set to 1, the track is used in the movie.

`trackUsageInPreview`
The track is used in the movie preview. If this flag is set to 1, the track is used in the preview.

`trackUsageInPoster`
The track is used in the movie poster. If this flag is set to 1, the track is used in the poster.

ERROR CODES

`invalidTrack`	–2009	This track is corrupted or invalid

ShowMoviePoster

You can use the `ShowMoviePoster` function to display a movie's poster. The movie poster uses the movie's matrix and display clipping characteristics.

```
pascal void ShowMoviePoster (Movie theMovie);
```

`theMovie` Specifies the movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

DESCRIPTION

The Movie Toolbox draws the movie poster once, in the movie's graphics world. This function works on active and inactive movies.

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid

SEE ALSO

You can set the poster's starting time with the `SetMoviePosterTime` function (described on page 2-118). You can set the position and size of the poster by calling the `SetPosterBox` function (described in the next section).

SetPosterBox

You can use the `SetPosterBox` function to set a poster's boundary rectangle. You define the poster's image by specifying a time in the movie (use the `SetMoviePosterTime` function, described on page 2-118). You specify the size and position of the poster image with this function.

```
pascal void SetPosterBox (Movie theMovie, const Rect *boxRect);
```

`theMovie` Specifies the movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

`boxRect` Contains a pointer to a rectangle. The Movie Toolbox sets the poster's boundary rectangle to the coordinates specified in the structure referred to by this parameter.

DESCRIPTION

If you do not specify a boundary rectangle for the poster, the Movie Toolbox uses the movie's matrix when it displays the poster.

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid
`invalidRect`	–2036	Specified rectangle has invalid coordinates

SEE ALSO

Your application can retrieve a poster's boundary rectangle by calling the `GetPosterBox` function, which is described in the next section.

GetPosterBox

The `GetPosterBox` function allows you to obtain a poster's boundary rectangle.

```
pascal void GetPosterBox (Movie theMovie, Rect *boxRect);
```

`theMovie` Specifies the movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

`boxRect` Contains a pointer to a rectangle. The Movie Toolbox returns the poster's boundary rectangle into the structure referred to by this parameter.

DESCRIPTION

When you call `GetPosterBox` without having called `SetPosterBox`, the current movie matrix is applied to the poster tracks to determine the poster box.

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid

SEE ALSO

You set the poster's boundary rectangle by calling the `SetPosterBox` function, which is described in the previous section.

SetMoviePosterTime

The `SetMoviePosterTime` function sets the poster time for the movie. Since a movie poster is a still frame, it is defined by a point in time within the movie. The poster's time is expressed in the movie's time coordinate system. Your application can retrieve a poster's time by calling the `GetMoviePosterTime` function, which is described in the next section.

```
pascal void SetMoviePosterTime (Movie theMovie,
                                TimeValue posterTime);
```

`theMovie` Specifies the movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

posterTime
Contains the starting time for the movie frame that contains the poster image.

ERROR CODES

invalidMovie	–2010	This movie is corrupted or invalid
invalidTime	–2015	This time value is invalid

SEE ALSO

Your application can set the poster's boundary rectangle by calling the SetPosterBox function, which is described on page 2-117.

GetMoviePosterTime

The GetMoviePosterTime function returns the poster's time in the movie. Since a movie poster has no duration, a poster is defined by a point in time within the movie. The time value returned is in the time coordinate system of the movie.

```
pascal TimeValue GetMoviePosterTime (Movie theMovie);
```

theMovie Specifies the movie for this operation. Your application obtains this movie identifier from such functions as NewMovie, NewMovieFromFile, and NewMovieFromHandle (described on page 2-92, page 2-88, and page 2-90, respectively).

DESCRIPTION

The GetMoviePosterTime function returns a time value. This time value contains the starting time for the movie frame that contains the movie poster image.

ERROR CODES

invalidMovie	–2010	This movie is corrupted or invalid

SEE ALSO

Your application can set a poster's time by calling the SetMoviePosterTime function, which is described in the previous section.

PlayMoviePreview

The `PlayMoviePreview` function plays a movie's preview.

```
pascal void PlayMoviePreview (Movie theMovie,
                              MoviePreviewCallOutProc callOutProc,
                              long refcon);
```

`theMovie` Specifies the movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

`callOutProc`
Contains a pointer to a movie callout function in your application. The Movie Toolbox calls this 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 this parameter to `nil`.

Your function should have the following form:

```
pascal Boolean MyCallOutProc (long refcon);
```

The `refCon` parameter contains the reference constant you specified when you called the `PlayMoviePreview` function.

Your function returns a Boolean value. The Movie Toolbox examines this value before continuing. If your function sets this value to `false`, the Movie Toolbox stops the preview and returns to your application. For details, see "Movie Callout Functions" on page 2-359.

Note that if you call the `GetMovieActiveSegment` function (described on page 2-137) from within your movie callout function, the Movie Toolbox will have changed the active movie segment to be the preview segment of the movie. The Movie Toolbox restores the active segment when the preview is done playing.

`refcon` Contains a reference constant for your function. The Movie Toolbox passes this value to your function.

DESCRIPTION

The `PlayMoviePreview` function sets the movie into preview mode, plays the movie preview, sets the movie back to normal playback mode, and returns to your application. The Movie Toolbox plays the preview in the movie's graphics world.

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid

SEE ALSO

Use the `SetMoviePreviewTime` function, described on page 2-122, to define the starting time and duration of the movie preview.

SetMoviePreviewMode

The `SetMoviePreviewMode` function allows your application to place a movie into and out of preview mode. When a movie is in preview mode, only those tracks identified as preview tracks are serviced. You specify how a track is used by calling the `SetTrackUsage` function, which is described on page 2-115.

```
pascal void SetMoviePreviewMode (Movie theMovie,
                                 Boolean usePreview);
```

`theMovie` Specifies the movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

`usePreview`
Specifies the movie's mode. Set this parameter to `true` to place the movie into preview mode. Set this parameter to `false` to place the movie into normal playback mode.

DESCRIPTION

When you place a movie into preview mode, the Movie Toolbox sets the active movie segment to be the preview segment of the movie. When you take a movie out of preview mode and place it back in normal playback mode, the toolbox sets the active movie segment to be the entire movie. For information about working with active movie segments, see "Enhancing Movie Playback Performance" beginning on page 2-134.

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid

GetMoviePreviewMode

The `GetMoviePreviewMode` function allows your application to determine whether a movie is in preview mode. If a movie is in preview mode, only the movie's preview can be displayed. Your application can place a movie into and out of preview mode by calling the `SetMoviePreviewMode` function, which is described in the previous section.

```
pascal Boolean GetMoviePreviewMode (Movie theMovie);
```

`theMovie` Specifies the movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

DESCRIPTION

The `GetMoviePreviewMode` function returns a Boolean value. If the movie is in preview mode, the function sets this return value to `true`. If the movie is in normal playback mode, the function sets this value to `false`.

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid

SetMoviePreviewTime

The `SetMoviePreviewTime` function allows your application to define the starting time and duration of the movie's preview. These time values are in the movie's time coordinate system.

```
pascal void SetMoviePreviewTime (Movie theMovie,
                                 TimeValue previewTime,
                                 TimeValue previewDuration);
```

`theMovie` Specifies the movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

`previewTime`
Contains a time value that specifies the preview's starting time.

`previewDuration`
Contains a time value that specifies the preview's duration.

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid
`invalidDuration`	–2014	This duration value is invalid
`invalidTime`	–2015	This time value is invalid

SEE ALSO

Your application can retrieve the starting time and duration of the preview with the `GetMoviePreviewTime` function, which is described in the next section.

GetMoviePreviewTime

The `GetMoviePreviewTime` function returns the starting time and duration of the movie's preview. These time values are expressed in the movie's time coordinate system.

```
pascal void GetMoviePreviewTime (Movie theMovie,
                                 TimeValue *previewTime,
                                 TimeValue *previewDuration);
```

`theMovie` Specifies the movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

`previewTime`
Contains a pointer to a time value. The Movie Toolbox places the preview's starting time into the field referred to by this parameter. If the movie does not have a preview, the Movie Toolbox sets this returned value to 0.

`previewDuration`
Contains a pointer to a time value. The Movie Toolbox places the preview's duration into the field referred to by this parameter. If the movie does not have a preview, the Movie Toolbox sets this returned value to 0.

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid

SEE ALSO

Your application sets the starting time and duration of the movie preview with the `SetMoviePreviewTime` function, which is described in the previous section.

Movies and Your Event Loop

In order for your movies to play, your application must grant time to the Movie Toolbox. You do this by calling the `MoviesTask` function from your main event loop. The `MoviesTask` function causes the Movie Toolbox to service all your active movies. You should call this function regularly so that your movie can play smoothly. You can use the `UpdateMovie` function to force your movie to be redrawn after it has been uncovered.

You may want your application to take a particular action when a movie is done playing. The Movie Toolbox provides the `IsMovieDone` function, which allows you to determine whether a movie is done playing. The Movie Toolbox also provides more sophisticated callback mechanisms, which are discussed in "Time Base Functions" beginning on page 2-315.

The Movie Toolbox provides two functions that allow your application to determine whether a specified point lies in either a movie or a track. Use the `PtInMovie` function with movies; use the `PtInTrack` function with tracks.

Your application can retrieve some status information about movies and tracks. Use the `GetMovieStatus` function to retrieve movie status; use the `GetTrackStatus` function to get track status.

MoviesTask

The `MoviesTask` function services active movies.

```
pascal void MoviesTask (Movie theMovie, long maxMilliSecToUse);
```

`theMovie` Specifies the movie for this operation. If you set this parameter to `nil`, the Movie Toolbox services all of your active movies. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

`maxMilliSecToUse`
Determines the maximum number of milliseconds that `MoviesTask` can work before returning. If this parameter is 0, `MoviesTask` services every active movie exactly once and then returns. If the parameter is nonzero, `MoviesTask` services as many movies as it can in the allotted time before returning.

Once the `MoviesTask` function starts servicing a movie, it cannot stop until it has completely met the requirements of the movie. Consequently, the `MoviesTask` function may execute for a longer time than that specified in `maxMilliSecToUse`. However, the function does not start servicing a new movie if the time specified by `maxMilliSecToUse` has elapsed.

The preferred way to use `MoviesTask` is to set the `maxMilliSecToUse` parameter to 0; however, if you just want to play one movie, you can call `MoviesTask` on that one.

If your rate is 0, `MoviesTask` draws that frame and no other.

DESCRIPTION

When servicing a movie, the Movie Toolbox performs the processing that is appropriate for the movie—displaying frames, playing sound, reading data from disk, or other tasks. The only time the Movie Toolbox actually draws a movie is during the operation of the `MoviesTask` function.

You should call `MoviesTask` as often as possible from your application's main event loop. Note that you should call this function after you have performed your own event processing.

The `MoviesTask` function services only active movies, and only enabled tracks within those active movies. Use the `SetMovieActive` function (described on page 2-145) and the `SetTrackEnabled` function (described on page 2-147) to enable and disable movies and tracks.

SPECIAL CONSIDERATIONS

Note that the `MoviesTask` function services only your movies. Your application must call the Event Manager's `WaitNextEvent` routine (or the Event Manager's `GetNextEvent` routine and the `SystemTask` routine) to give other applications the opportunity to call `MoviesTask` for their movies. For details on `WaitNextEvent`, `GetNextEvent`, and `SystemTask`, see *Inside Macintosh: Macintosh Toolbox Essentials*.

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid

IsMovieDone

Your application may wish to take a particular action when a movie is done playing. The `IsMovieDone` function allows you to determine if a particular movie has completely finished playing.

```
pascal Boolean IsMovieDone (Movie theMovie);
```

`theMovie` Specifies the movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

DESCRIPTION

The `IsMovieDone` function returns `true` if the specified movie has finished playing; otherwise it returns `false`. A movie with a positive rate (playing forward) is considered done when its movie time reaches the movie end time. Conversely, a movie with a negative rate (playing backward) is considered done when its movie time reaches the movie start time.

If your application has changed the movie's active segment, the status returned by the `IsMovieDone` function is relative to the active segment, rather than to the entire movie. You can use the `SetMovieActiveSegment` function (described on page 2-136) to change a movie's active segment.

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid

UpdateMovie

The `UpdateMovie` function allows your application to ensure that the Movie Toolbox properly displays your movie after it has been uncovered.

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

```
pascal OSErr UpdateMovie (Movie theMovie);
```

`theMovie` Specifies the movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

DESCRIPTION

The `UpdateMovie` function does not actually update the movie's graphics world. Rather, the function invalidates the movie's display state so that the Movie Toolbox redraws the movie the next time you call the `MoviesTask` function. If you need to force a movie to be redrawn outside of a Window Manager update sequence, your application can call `UpdateMovie` and then call the `MoviesTask` function (described on page 2-124) to service the movie.

The Movie Toolbox determines the portion of the screen to update by examining the graphics port's visible region.

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid

SEE ALSO

For sample code that uses the `UpdateMovie` function in a Window Manager update sequence, see Listing 2-13 on page 2-63.

PtInMovie

The `PtInMovie` function allows your application to determine whether a specified point lies in the region defined by a movie's final display boundary region after it has been clipped by the movie's display clipping region. This function is accurate at the current movie time.

```
pascal Boolean PtInMovie (Movie theMovie, Point pt);
```

`theMovie`	Specifies the movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).
`pt`	Specifies the point to be checked. This point must be expressed in the movie's local display coordinate system.

DESCRIPTION

The `PtInMovie` function returns a Boolean value. The function sets this value to `true` if the point lies in the movie's display space.

SPECIAL CONSIDERATIONS

The region that `PtInMovie` checks for is different from the movie box.

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid

SEE ALSO

To find out if a point lies in the region defined by a track's display boundary region after it has been clipped by a movie's final display clipping region, you use the `PtInTrack` function. See the next section for details.

PtInTrack

The `PtInTrack` function allows your application to determine whether a specified point lies in the region defined by a track's display boundary region after it has been clipped by the movie's final display clipping region. This function is accurate at the current movie time.

```
pascal Boolean PtInTrack (Track theTrack, Point pt);
```

`theTrack` Specifies the track for this operation. Your application obtains this track identifier from such Movie Toolbox functions as `NewMovieTrack` and `GetMovieTrack` (described on page 2-151 and page 2-204, respectively).

`pt` Specifies the point to be checked. This point must be expressed in the local display coordinate system of the movie that contains the track.

DESCRIPTION

The `PtInTrack` function returns a Boolean value. The function sets this value to `true` if the point lies in the track's display space.

SPECIAL CONSIDERATIONS

The region that `PtInTrack` checks for is different from the movie box.

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid

SEE ALSO

To find out if a point lies within the region defined by a movie's final display boundary region after it has been clipped by the movie's display clipping region, you can use the `PtInMovie` function, which is described in the previous section.

GetMovieStatus

The `GetMovieStatus` function searches for errors in all the enabled tracks of the movie. This function returns information about errors that are encountered during the processing associated with the `MoviesTask` function (described on page 2-124). These errors typically reflect playback problems, such as low-memory conditions.

```
pascal ComponentResult GetMovieStatus (Movie theMovie,
                                       Track *firstProblemTrack);
```

`theMovie` Specifies the movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

`firstProblemTrack`
Contains a pointer to a track identifier. The Movie Toolbox places the identifier for the first track that is found to contain an error into the field referred to by this parameter. If you do not want to receive the track identifier, set this parameter to `nil`.

DESCRIPTION

The `GetMovieStatus` function returns the error from the first problem track. If the component does not find any errors, the result is set to `noErr`.

ERROR CODES

Any Movie Toolbox result code (see "Summary of the Movie Toolbox" at the end of this chapter)

GetTrackStatus

The `GetTrackStatus` function returns the value of the last error the media encountered while playing a specified track. This function returns information about errors that are encountered during the processing associated with the `MoviesTask` function (described on page 2-124). These errors typically reflect playback problems, such as low-memory conditions.

The media clears this error code when it detects that the error has been corrected.

```
pascal ComponentResult GetTrackStatus (Track theTrack);
```

`theTrack` Specifies the track for this operation. Your application obtains this track identifier from the `GetMovieStatus` function, described in the previous section.

DESCRIPTION

The `GetTrackStatus` function returns the last error encountered for the specified track. If the component does not find any errors, the result is set to `noErr`.

ERROR CODES

Any Movie Toolbox result code (see "Summary of the Movie Toolbox" at the end of this chapter)

Preferred Movie Settings

Every movie has default, or preferred, settings for playback rate and volume. These settings are stored with the movie in its movie file. The Movie Toolbox provides functions that allow your application to manipulate these default settings.

You can use the `GetMoviePreferredRate` and `SetMoviePreferredRate` functions to work with a movie's default playback rate. You can use the `GetMoviePreferredVolume` and `SetMoviePreferredVolume` functions to work with the default sound volume of a movie.

You can use the `SetMovieRate` function to change a movie's playback rate—see "Working with Movie Time" beginning on page 2-184 for a complete description of this function. The Movie Toolbox also provides a number of functions that allow you to change other settings when you play a movie. These functions are discussed in "Functions That Modify Movie Properties" beginning on page 2-157.

SetMoviePreferredRate

The `SetMoviePreferredRate` function allows your application to specify a movie's default playback rate.

```
pascal void SetMoviePreferredRate (Movie theMovie, Fixed rate);
```

`theMovie` Specifies the movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

`rate` Specifies the new movie rate as a 32-bit, fixed-point number. Positive integers indicate forward rates and negative integers indicate reverse rates.

DESCRIPTION

The default playback rate is the rate that the `StartMovie` function (described on page 2-111) uses when it starts playing a movie. The default preferred rate of a movie is set to 1.0 (the `kFix1` constant) when the movie is created.

SPECIAL CONSIDERATIONS

Do not set the preferred rate to 0.

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid

SEE ALSO

Your application can obtain the preferred playback rate by calling the `GetMoviePreferredRate` function, which is described in the next section.

You can set the current playback rate of a movie by calling the `SetMovieRate` function, which is described on page 2-187.

GetMoviePreferredRate

The `GetMoviePreferredRate` function returns a movie's default playback rate. This is the rate that the `StartMovie` function uses when it starts playing a movie.

```
pascal Fixed GetMoviePreferredRate (Movie theMovie);
```

`theMovie` Specifies the movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

DESCRIPTION

The `GetMoviePreferredRate` function returns the default movie rate as a 32-bit, fixed-point number. Positive integers indicate forward rates and negative integers indicate reverse rates.

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid

SEE ALSO

Your application can change the preferred playback rate by calling the `SetMoviePreferredRate` function, which is described in the previous section. You can change the current playback rate of a movie by calling the `SetMovieRate` function, which is described on page 2-187.

SetMoviePreferredVolume

The `SetMoviePreferredVolume` function allows your application to set a movie's preferred volume setting.

```
pascal void SetMoviePreferredVolume (Movie theMovie,
                                     short volume);
```

`theMovie` Specifies the movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

`volume` Specifies the preferred volume setting of the movie. The `volume` parameter must contain a 16-bit, fixed-point number that contains the movie's default volume. The high-order 8 bits contain the integer part of the value; the low-order 8 bits contain the fractional part. Volume values range from –1.0 to 1.0. Negative values play no sound but preserve the absolute value of the volume setting. You may find the following constants useful:

`kFullVolume`
Sets the movie to full volume (constant value is 1.0).

`kNoVolume`
Sets the movie to no volume (constant value is 0.0).

DESCRIPTION

Your application can obtain the preferred volume setting by calling the `GetMoviePreferredVolume` function, which is described in the next section. You can change a movie's current volume by calling the `SetMovieVolume` function, which is described on page 2-182.

A movie's tracks may have their own volume settings. Use the `SetTrackVolume` function, described on page 2-183, to set the volume of an individual track. 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 Operating System's `GetSoundVol` routine (described in *Inside Macintosh: More Macintosh Toolbox*). Thus, the user can control the overall volume from the Sound control panel.

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid

SEE ALSO

When a movie is loaded, the current setting is set to preferred volume. The `StartMovie` function (described on page 2-111) uses this volume setting when it starts playing a movie.

GetMoviePreferredVolume

The `GetMoviePreferredVolume` function returns a movie's preferred volume setting.

```
pascal short GetMoviePreferredVolume (Movie theMovie);
```

`theMovie` Specifies the movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

DESCRIPTION

The `GetMoviePreferredVolume` function returns a 16-bit, fixed-point number that contains the movie's default volume. The high-order 8 bits contain the integer part of the value; the low-order 8 bits contain the fractional part. Volume values range from 0.0 to 1.0.

You can change a movie's current volume by calling the `SetMovieVolume` function, which is described on page 2-182.

A movie's tracks have their own volume settings. Use the `SetTrackVolume` function, described on page 2-183, to set the volume of an individual track. 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 Operating System's `GetSoundVol` routine (described in *Inside Macintosh: More Macintosh Toolbox*). Thus, the user can control the overall volume from the Sound control panel.

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid

SEE ALSO

When a movie is loaded, the current setting is set to preferred volume. The `StartMovie` function (described on page 2-111) uses this volume setting when it starts playing a movie.

Enhancing Movie Playback Performance

There are circumstances in which an application needs to optimize the performance of a movie or a portion of a movie. The Movie Toolbox provides several functions to help in this process.

The first step you can take to enhance movie playback performance is to allow the Movie Toolbox to **preroll** the movie. When the toolbox prerolls a movie, it informs the media handlers that the movie is about to play. The media handlers can then load the appropriate movie data. In this manner, the movie can play smoothly from the start. Use the `PrerollMovie` function to preroll a movie.

The next performance enhancement technique is to load portions of a movie, track, or media into memory, thus reducing or eliminating disk access during playback. Loading the movie into RAM provides most noticeable performance improvements when there is a lot of random access involved in the playback process and the entire movie fits into available memory. Use the `LoadMovieIntoRam`, `LoadTrackIntoRam`, and `LoadMediaIntoRam` functions to copy all or part of a movie into memory.

Note
The `LoadMovieIntoRam`, `LoadTrackIntoRam`, and `LoadMediaIntoRam` functions load tracks into memory in a time-slice order so that, if a function fails because it is out of memory, all tracks are left loaded to about the same point in time. u

You can influence the temporal accuracy, and therefore the speed, with which the Movie Toolbox tries to display a movie by calling either the `SetMoviePlayHints` or `SetMediaPlayHints` function.

For each movie currently in use, the Movie Toolbox maintains an active movie segment. The active movie segment is the part of the movie that your application is interested in playing. By default, the active movie segment is set to be the entire movie. You may wish to change this to be some segment of the movie—for example, if you wish to play a user's selection repeatedly. By setting the active movie segment you guarantee that the Movie Toolbox uses no samples from outside of that range while playing the movie. Use the `SetMovieActiveSegment` and `GetMovieActiveSegment` functions to work with the active segment.

Some movies contain very few key frames and a great number of frame differences. These movies play back very well because they have a lower data rate. Unfortunately, this makes random access operations, such as scrubbing, on a movie difficult. In such movies, random access is difficult.

To improve random access performance of movies with few key frames and many frame differences, shadow sync samples may be added. **Shadow sync samples** are self-contained samples that are alternates for already existing frame difference samples.

During certain random access operations, a shadow sync sample is used instead of a normal key frame, which may be very far away from the desired frame.

The Movie Toolbox provides two functions to let you create just such an association between a frame difference sample and a sync sample. `SetMediaShadowSync` establishes a shadow sync sample for a media. You can use `GetMediaShadowSync` to find out if a particular frame difference sample has a shadow sync sample.

PrerollMovie

The `PrerollMovie` function allows your application to prepare a portion of a movie for playback.

```
pascal OSErr PrerollMovie (Movie theMovie, TimeValue time,
                           Fixed Rate);
```

`theMovie` Specifies the movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

`time` Contains the starting time of the movie segment to play.

`Rate` Specifies the rate at which you anticipate playing the movie. You specify the movie rate as a 32-bit, fixed-point number. Positive integers indicate forward rates and negative integers indicate reverse rates.

DESCRIPTION

When your application calls the `PrerollMovie` function, the Movie Toolbox tells the appropriate media handlers to prepare to play the movie. The media handlers may then load the movie data and perform any other necessary preparations to play the movie, such as allocating sound channels and starting up image-decompression sequences. In this manner, you can eliminate playback stutter when the movie starts playing.

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid
`invalidTime`	–2015	This time value is invalid

SetMovieActiveSegment

You can use the `SetMovieActiveSegment` function to define a movie's active segment. Your application defines the active segment by specifying the starting time and duration of the segment. These values must be expressed in the movie's time coordinate system. By default, the entire movie is active.

```
pascal void SetMovieActiveSegment (Movie theMovie,
                                   TimeValue startTime,
                                   TimeValue duration);
```

`theMovie` Specifies the movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

`startTime` Contains a time value specifying the starting point of the active segment. Set this parameter to –1 to make the entire movie active. In this case, the `SetMovieActiveSegment` function ignores the `duration` parameter.

`duration` Contains a time value that specifies the duration of the active segment. If you are making the entire movie active (by setting the `startTime` parameter to –1), the Movie Toolbox ignores this parameter.

DESCRIPTION

Your application can retrieve the information that defines a movie's active segment by calling the `GetMovieActiveSegment` function, which is described in the next section.

SPECIAL CONSIDERATIONS

Note that placing a movie into preview mode destroys the movie's active segment. You use the `SetMoviePreviewMode` function, described on page 2-121, to control preview mode.

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid
`invalidDuration`	–2014	This duration value is invalid
`invalidTime`	–2015	This time value is invalid

GetMovieActiveSegment

Use the `GetMovieActiveSegment` function to determine what portion of a movie is currently active for playing.

```
pascal void GetMovieActiveSegment (Movie theMovie,
                                   TimeValue *startTime,
                                   TimeValue *duration);
```

`theMovie` Specifies the movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

`startTime` Contains a pointer to a time value. The `GetMovieActiveSegment` function places the starting time of the active segment into the field referred to by this parameter. If the returned time value is set to –1, the entire movie is active. In this case, the Movie Toolbox does not return any duration information via the `duration` parameter.

`duration` Contains a pointer to a time value. The `GetMovieActiveSegment` function places the duration of the active movie segment into the field referred to by this parameter. If the entire movie is active (the returned starting time is set to –1), the Movie Toolbox does not return any duration information.

DESCRIPTION

Your application can set the active segment by calling the `SetMovieActiveSegment` function, which is described in the previous section.

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid

SetMoviePlayHints

The `SetMoviePlayHints` function allows your application to provide information to the Movie Toolbox that can influence movie playback. This function accepts a flag in which you specify optimizations that the Movie Toolbox can use during movie playback. These optimizations apply to all of the media structures used by the movie.

```
pascal void SetMoviePlayHints (Movie theMovie, long flags,
                               long flagsMask);
```

`theMovie` Specifies the movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

`flags` Specifies the optimizations that can be used with this movie. Each bit in the `flags` parameter corresponds to a specific optimization. The following flag is defined (be sure to set unused flags to 0):

`hintsScrubMode`
Indicates that the Movie Toolbox can prefer to display key frames when the movie is repositioned. This optimization is used only when a movie's rate is set to 0. If you set this flag to 1, the Movie Toolbox is free to display the nearest key frame when you set the movie's current time; the Movie Toolbox then moves to the appropriate frame as time permits. If you set this flag to 0, the Movie Toolbox displays the frame that corresponds to the new current time, even if that frame is not a key frame.

By displaying key frames first, the Movie Toolbox can display data from temporally compressed movies much more quickly in response to changes to the movie's current time. This, in turn, can improve the liveliness of a movie control. For example, if the user is positioning in a stopped movie, the Movie Toolbox can display a key frame that corresponds to the new position without having to build up the image offscreen. In this manner, the user gets quicker feedback from your application.

`hintsUseSoundInterp`
Turns on sound interpolation—that is, tells the Sound Manager to use sound interpolation when playing back sound. In certain situations, this improves the sound quality to 11 kHz.

`hintsAllowInterlace`
Tells the Image Compression Manager to use the interlace option for image compressor and decompressor components. For more information, see *Inside Macintosh: QuickTime Components*.

`flagsMask` Indicates which flags in the `flags` parameter are to be considered in this operation. For each bit in the `flags` parameter that you want the Movie Toolbox to consider, you must set the corresponding bit in the `flagsMask` parameter to 1. Set unused flags to 0. This allows you to work with a single optimization without altering the settings of other flags.

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid

SetMediaPlayHints

The `SetMediaPlayHints` function allows your application to provide information to the Movie Toolbox that can influence playback of a single media. This function accepts a flag in which you specify optimizations that the Movie Toolbox can use during movie playback. These optimizations apply to only the specified media.

```
pascal void SetMediaPlayHints (Media theMedia, long flags,
                               long flagsMask);
```

`theMedia` Specifies the media for this operation. Your application obtains this media identifier from such Movie Toolbox functions as `NewTrackMedia` and `GetTrackMedia` (described on page 2-153 and page 2-206, respectively).

`flags` Specifies the optimizations that can be used with this media. Each bit in the `flags` parameter corresponds to a specific optimization. The following flag is defined (be sure to set unused flags to 0):

`hintsScrubMode`

Indicates that the Movie Toolbox can prefer to display key frames when the movie that uses this media is repositioned. This optimization is used only when a movie's rate is set to 0. If you set this flag to 1, the Movie Toolbox is free to display the nearest key frame when you set the movie's current time; the Movie Toolbox then moves to the appropriate frame as time permits. If you set this flag to 0, the Movie Toolbox displays the frame that corresponds to the new current time, even if that frame is not a key frame.

By displaying key frames first, the Movie Toolbox can display data from temporally compressed movies much more quickly in response to changes to the movie's current time. This, in turn, can improve the liveliness of a movie control. For example, if the user is positioning in a stopped movie, the Movie Toolbox can display a key frame that corresponds to the new position without having to build up the image offscreen. In this manner, the user gets quicker feedback from your application.

`hintsUseSoundInterp`

Turns on sound interpolation—that is, tells the Sound Manager to use sound interpolation when playing back sound. In certain situations, this improves the sound quality to 11 kHz.

`hintsAllowInterlace`

Tells the Image Compression Manager to use the interlace option for image compressor and decompressor components. For more information, see *Inside Macintosh: QuickTime Components*.

`flagsMask` Indicates which flags in the `flags` parameter are to be considered in this operation. For each bit in the `flags` parameter that you want the Movie Toolbox to consider, you must set the corresponding bit in the `flagsMask` parameter to 1. Set unused flags to 0. This allows you to work with a single optimization without altering the settings of other flags.

ERROR CODES

`invalidMedia`	–2008	This media is corrupted or invalid

SEE ALSO

To set optimizations for all of a movie's media structures, use the `SetMoviePlayHints` function, which is described in the previous section.

LoadMovieIntoRam

The `LoadMovieIntoRam` function loads a movie's data into memory. If the movie does not fit, the function returns an error.

```
pascal OSErr LoadMovieIntoRam (Movie theMovie, TimeValue time,
                               TimeValue duration,
                               long flags);
```

`theMovie` Specifies the movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

`time` Allows you to specify a portion of the movie to load. The `time` parameter contains the starting time of the movie segment to load. The `duration` parameter specifies the length of the segment to load.

`duration` Allows you to specify a portion of the movie to load. The `time` parameter contains the starting time of the movie segment to load. The `duration` parameter specifies the length of the segment to load. You can use the `GetMovieDuration` function (described on page 2-185) to determine the length of the entire movie. Note that the Movie Toolbox may load more data than you specify due to the way the data is loaded.

`flags` Gives you explicit control over what is loaded into memory and how long to keep it around. The following constants are provided. You can set these flags in any combination that makes sense for you.

`keepInRam`
Renders all data loaded with this flag set as nonpurgeable. Nonpurgeable data is not released from memory until you request it explicitly. This practice can fill up your heap very quickly. Exercise caution.

`unkeepInRam`
: Renders all indicated data purgeable. The data is not necessarily released from memory immediately, however. Information about whether a chunk can be purged is maintained internally by a single bit. This means there is no counter. Therefore, if you care very much about the data, you have to work very hard and use the edit list meticulously.

`flushFromRam`
: Purges all indicated data from memory, unless it is currently in use by a media handler (for example, if it is still drawing frames from the requested times). This flag makes the memory available for purging, and then performs the purge. You may want to use this option if you are particularly low on memory.

`loadForwardTrackEdits`
: In some cases, an edited movie plays back much more smoothly if the data around edits is already in RAM. By setting either this flag or the `lookBackwardTrackEdits` flag, you can load only the data around edits. The Movie Toolbox walks through the edits and decides the right amount of data to load for you. If you are going to play the movie forward, set only the `loadForwardTrackEdits` flag. If you are going to play in both directions, or you don't know which direction, set both flags.

`loadBackwardTrackEdits`
: In some cases, an edited movie plays back much more smoothly if the data around edits is already in RAM. By setting either this flag or `lookForwardTrackEdits`, you can load only the data around edits. The Movie Toolbox walks through the edits and decides the right amount of data to load for you. If you are going to play the movie only backward, set the `loadBackwardTrackEdits` flag. If you are going to play in both directions, or you don't know which direction, set both flags.

DESCRIPTION

If `LoadMovieIntoRam` fails because it was out of memory, no data is purged.

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid
`invalidDuration`	–2014	This duration value is invalid
`invalidTime`	–2015	This time value is invalid
`progressProcAborted`	–2019	Your progress function returned an error

File Manager errors

Memory Manager errors

LoadTrackIntoRam

The `LoadTrackIntoRam` function loads a track's data into memory. If the track does not fit, the function returns an error.

```
pascal OSErr LoadTrackIntoRam (Track theTrack, TimeValue time,
                               TimeValue duration, long flags);
```

`theTrack` Specifies the track for this operation. Your application obtains this track identifier from such Movie Toolbox functions as `NewMovieTrack` and `GetMovieTrack` (described on page 2-151 and page 2-204, respectively).

`time` Allows you to specify a portion of the track to load. The `time` parameter contains the starting time of the track segment to load. The `duration` parameter specifies the length of the segment to load. You must specify this time value in the movie's time coordinate system.

`duration` Allows you to specify a portion of the track to load. The `time` parameter contains the starting time of the track segment to load. The `duration` parameter specifies the length of the segment to load. You can use the `GetTrackDuration` function (described on page 2-191) to determine the length of the entire movie. Note that the media handler may load more data than you specify.

`flags` Gives you explicit control over what is loaded into memory and how long to keep it around. The following constants are provided:

```
enum
{
   keepInRam = 1<<0,
   unkeepInRam = 1<<1,
   flushFromRam = 1<<2,
   loadForwardTrackEdits = 1<<3,
   loadBackwardTrackEdits = 1<<4
};
```

You can set these flags in any combination that makes sense. For descriptions of the individual flag constants, see the description of the `LoadMovieIntoRam` function on page 2-140.

ERROR CODES

`invalidTrack`	–2009	This track is corrupted or invalid
`invalidDuration`	–2014	This duration value is invalid
`invalidTime`	–2015	This time value is invalid
`progressProcAborted`	–2019	Your progress function returned an error

File Manager errors

Memory Manager errors

LoadMediaIntoRam

The `LoadMediaIntoRam` function loads a media's data into memory.

The exact behavior of `LoadMediaIntoRam` is dependent on the media handler.

```
pascal OSErr LoadMediaIntoRam (Media theMedia, TimeValue time,
                               TimeValue duration, long flags);
```

`theMedia` Specifies the media for this operation. Your application obtains this media identifier from such Movie Toolbox functions as `NewTrackMedia` and `GetTrackMedia` (described on page 2-153 and page 2-206, respectively).

`time` Allows you to specify a portion of the media to load. The `time` parameter contains the starting time of the media segment to load. The `duration` parameter specifies the length of the segment to load. This time value must be expressed in the media's time coordinate system.

`duration` Allows you to specify a portion of the media to load. The `time` parameter contains the starting time of the media segment to load. The `duration` parameter specifies the length of the segment to load. You can use the `GetMediaDuration` function (described on page 2-194) to determine the length of the entire media. Note that the media handler may load more data than you specify if the media data was added in larger pieces.

`flags` Gives you explicit control over what is loaded into memory and how long to keep it around. The following constants are provided:

```
enum
{
   keepInRam = 1<<0,
   unkeepInRam = 1<<1,
   flushFromRam = 1<<2,
};
```

You can set these flags in any combination that makes sense. For descriptions of the individual flag constants, see the description of the `LoadMovieIntoRam` function on page 2-140.

DESCRIPTION

If the `LoadMediaIntoRam` function fails because it is out of memory, no data is purged.

ERROR CODES

`invalidMedia`	–2008	This media is corrupted or invalid
`invalidDuration`	–2014	This duration value is invalid
`invalidTime`	–2015	This time value is invalid
`progressProcAborted`	–2019	Your progress function returned an error

File Manager errors

Memory Manager errors

SetMediaShadowSync

The `SetMediaShadowSync` function creates an association between the indicated frame difference sample and a specified self-contained sample in a given media. This association makes the self-contained sample a shadow sync sample for the frame difference sample.

```
pascal OSErr SetMediaShadowSync (Media theMedia,
                                 long frameDiffSampleNum,
                                 long syncSampleNum);
```

`theMedia` The media in which the shadow sync is to be created.

`frameDiffSampleNum`
Specifies a frame difference sample. The sample number is obtained from the `MediaTimeToSampleNum` function.

`syncSampleNum`
Specifies a shadow sync sample. The sample number is obtained from the `MediaTimeToSampleNum` function.

DESCRIPTION

Note that the association established is between sample numbers—not sample times.

SPECIAL CONSIDERATIONS

Shadow sync samples should not be part of a track. You should not call `InsertMediaIntoTrack` on these media samples. Typically, you add shadow sync samples after a media is completely created. Shadow sync samples are not maintained when editing or flattening movies.

ERROR CODES

Memory Manager errors

GetMediaShadowSync

The `GetMediaShadowSync` function returns the sample number of the shadow sync associated with a given frame difference sample number.

```
pascal OSErr GetMediaShadowSync (Media theMedia,
                                 long frameDiffSampleNum,
                                 long *syncSampleNum);
```

`theMedia` Indicates the media in which the shadow sync sample has been established and the shadow sync number is to be obtained.

`frameDiffSampleNum`
Specifies the frame difference sample number associated with the desired shadow sync sample number.

`syncSampleNum`
Contains a pointer to the sample number of the shadow sync. If the `frameDiffSample` parameter does not have a shadow sync, 0 is returned in the `syncSampleNum` parameter.

ERROR CODES

Memory Manager errors

Disabling Movies and Tracks

The Movie Toolbox services only movies and tracks that are active. This section describes functions that allow your application to enable and disable tracks and movies.

You can use the `SetMovieActive` function to activate and deactivate a movie. Use the `GetMovieActive` function to determine whether a movie is active.

Similarly, your application can use the `SetTrackEnabled` function to enable and disable a track. Use the `GetTrackEnabled` function to determine whether a track is enabled. The Movie Toolbox also allows you to assign alternate tracks based on language or quality criteria. Functions that work with alternate tracks are discussed in "Working With Alternate Tracks" beginning on page 2-207.

SetMovieActive

The `SetMovieActive` function allows your application to activate and deactivate a movie.

```
pascal void SetMovieActive (Movie theMovie, Boolean active);
```

`theMovie` Specifies the movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

`active` Activates or deactivates the movie. Set this parameter to `true` to activate the movie; set this parameter to `false` to deactivate the movie.

SPECIAL CONSIDERATIONS

The Movie Toolbox services only active movies. When you deactivate a movie, the Movie Toolbox may release system resources required by the movie, such as sound hardware, open files, and allocated memory. Unless you set the `newMovieActive` flag when creating a movie, you should call `SetMovieActive` before playing a movie.

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid

SEE ALSO

You can determine whether a movie is active by calling the `GetMovieActive` function, which is described in the next section.

GetMovieActive

The `GetMovieActive` function allows your application to determine whether a movie is currently active. The Movie Toolbox services only active movies.

```
pascal Boolean GetMovieActive (Movie theMovie);
```

`theMovie` Specifies the movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

DESCRIPTION

The `GetMovieActive` function returns a Boolean value. The function sets this value to `true` if the movie is active and `false` if the movie is not active.

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid

SEE ALSO

You can make a movie active by calling the `SetMovieActive` function, which is described in the previous section.

SetTrackEnabled

The `SetTrackEnabled` function allows your application to enable and disable a track. The Movie Toolbox services only enabled tracks.

```
pascal void SetTrackEnabled (Track theTrack, Boolean isEnabled);
```

`theTrack` Specifies the track for this operation. Your application obtains this track identifier from such Movie Toolbox functions as `NewMovieTrack` and `GetMovieTrack` (described on page 2-151 and page 2-204, respectively).

`isEnabled` Enables or disables the track. Set this parameter to `true` to enable the track. Set this parameter to `false` to disable the track.

SPECIAL CONSIDERATIONS

When you disable a track, the Movie Toolbox may release system resources that are used by the track, including allocated memory.

ERROR CODES

`invalidTrack`	–2009	This track is corrupted or invalid

SEE ALSO

You can determine whether a track is enabled by calling the `GetTrackEnabled` function, which is described in the next section.

GetTrackEnabled

The `GetTrackEnabled` function allows your application to determine whether a track is currently enabled. The Movie Toolbox services only enabled tracks.

```
pascal Boolean GetTrackEnabled (Track theTrack);
```

`theTrack` Specifies the track for this operation. Your application obtains this track identifier from such Movie Toolbox functions as `NewMovieTrack` and `GetMovieTrack` (described on page 2-151 and page 2-204, respectively).

DESCRIPTION

The `GetTrackEnabled` function returns a Boolean value. The function sets this value to `true` if the track is enabled and `false` if the track is disabled.

ERROR CODES

`invalidTrack`	–2009	This track is corrupted or invalid

SEE ALSO

You can enable a track by calling the `SetTrackEnabled` function, which is described in the previous section.

Generating Pictures From Movies

The Movie Toolbox provides a set of functions that allow your application to create QuickDraw pictures from movies, tracks, and posters. This section discusses those functions.

You can use the `GetMoviePict` function to create a picture from a movie or its preview; you can use the `GetTrackPict` function to create a picture from a track. The `GetMoviePosterPict` function lets you create a picture that contains a movie's poster. If a movie or track has no spatial representation, the returned picture is empty—that is, the upper-left and lower-right coordinates are equal.

GetMoviePict

The `GetMoviePict` function creates a picture from the specified movie at the specified time. This function uses only those movie tracks that are currently enabled and would therefore be used in playback. Your application may call this function even if the movie is inactive.

```
pascal PicHandle GetMoviePict (Movie theMovie, TimeValue time);
```

`theMovie`	Specifies the movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).
`time`	Specifies the movie image for the picture. The `time` parameter contains the time from which the image is taken.

DESCRIPTION

The `GetMoviePict` function returns a handle to the picture. Your application must dispose of this picture handle by calling QuickDraw's `KillPicture` routine. If the function could not create the picture, the returned handle is set to `nil`.

SPECIAL CONSIDERATIONS

You can use the `GetMoviePict` function to create a picture. If the movie contains compressed data, the picture created by this function may also contain compressed data that cannot be displayed without QuickTime.

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid
`invalidTime`	–2015	This time value is invalid

Image Compression Manager errors
Memory Manager errors

SEE ALSO

If you want to create a picture from a movie's preview, put the movie into preview mode by calling the `SetMoviePreviewMode` function (described on page 2-121), and then call the `GetMoviePict` function.

GetMoviePosterPict

The `GetMoviePosterPict` function creates a picture that contains a movie's poster.

```
pascal PicHandle GetMoviePosterPict (Movie theMovie);
```

`theMovie` Specifies the movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

DESCRIPTION

The `GetMoviePosterPict` function returns a handle to the picture. Your application must dispose of this picture handle by calling QuickDraw's `KillPicture` routine. If the function could not create the picture, the returned handle is set to `nil`.

SPECIAL CONSIDERATIONS

If you have not assigned a poster time for the movie, the Movie Toolbox creates the poster from the movie image that corresponds to a time value of 0.

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid

Image Compression Manager errors
Memory Manager errors

GetTrackPict

The `GetTrackPict` function creates a QuickDraw picture from the specified track at the specified time. This function is similar to the `GetMoviePict` function (described on page 2-148), except that `GetTrackPict` uses only the specified track to create the picture.

```
pascal PicHandle GetTrackPict (Track theTrack, TimeValue time);
```

`theTrack` Specifies the track for this operation. Your application obtains this track identifier from such Movie Toolbox functions as `NewMovieTrack` and `GetMovieTrack` (described on page 2-151 and page 2-204, respectively).

`time` Specifies the track image for the picture. The `time` parameter contains the time from which the image is taken.

DESCRIPTION

The `GetTrackPict` function returns a handle to the picture. Your application must dispose of this picture handle by calling QuickDraw's `KillPicture` routine. If the function could not create the picture, the returned handle is set to `nil`.

SPECIAL CONSIDERATIONS

You can specify a disabled track. If the track contains compressed data, the picture created by this function may also contain compressed data that cannot be displayed without QuickTime.

ERROR CODES

`invalidTrack`	–2009	This track is corrupted or invalid
`invalidTime`	–2015	This time value is invalid

Image Compression Manager errors
Memory Manager errors

Creating Tracks and Media Structures

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.

NewMovieTrack

You can create movie tracks by calling the `NewMovieTrack` function. Immediately after creating a new track, you should call the `NewTrackMedia` function to create a media for the track—a track without a media is of no use.

Note that when you add a track to a movie, the Movie Toolbox automatically adjusts the display rectangle of the movie. You may want to detect these changes by calling the `GetMovieBox` function (described on page 2-162) so that you can adjust the size of the movie's display window.

```
pascal Track NewMovieTrack (Movie theMovie, Fixed width,
                            Fixed height, short trackVolume);
```

`theMovie` Specifies the movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

`width` Specifies a fixed number denoting the display width of the track, in pixels. Along with the `height` parameter, this parameter defines the track's display rectangle.

`height` Specifies a fixed number denoting the display height of the track, in pixels.

Together, the `height` and `width` parameters define the track's display rectangle. The upper-left corner of this rectangle lies at (0,0) in the movie's rectangle. The `height` and `width` parameters therefore establish the lower-right corner of the track's display rectangle. If you are creating a track that is not displayed, such as a sound track, set the `height` and `width` parameters to 0.

`trackVolume`

Specifies the volume setting of the track as a 16-bit, fixed-point number. The high-order 8 bits specify the integer portion; the low-order 8 bits specify the fractional part. Volume values range from –1.0 to 1.0. Negative values play no sound but preserve the absolute value of the volume setting. Set this parameter to `kFullVolume` to play the track at its full, natural volume. Set this parameter to `kNoVolume` to set the volume to 0.

`kFullVolume`

Sets the track to full volume (constant value is 1.0).

`kNoVolume`

Sets the track to no volume (constant value is 0.0).

DESCRIPTION

The `NewMovieTrack` function returns a track identifier. If the function cannot create the track, it sets the returned identifier to `nil`.

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid

Memory Manager errors

DisposeMovieTrack

The `DisposeMovieTrack` function removes a track from a movie.

```
pascal void DisposeMovieTrack (Track theTrack);
```

`theTrack` Specifies the track for this operation. Your application obtains this track identifier from such Movie Toolbox functions as `NewMovieTrack` and `GetMovieTrack` (described on page 2-151 and page 2-204, respectively).

DESCRIPTION

When you remove a track from a movie, the Movie Toolbox also removes the corresponding media from the movie.

SPECIAL CONSIDERATIONS

Your application should not call this function as part of the process of disposing of a movie. When you dispose of a movie by calling the `DisposeMovie` function (described on page 2-96), the Movie Toolbox disposes of all the movie's tracks and their associated media structures.

ERROR CODES

`invalidTrack`	–2009	This track is corrupted or invalid
`trackNotInMovie`	–2030	This track is not in this movie

NewTrackMedia

After you have created a new track, you can create a media for the track by calling the `NewTrackMedia` function. The media refers to the actual data samples used by the track.

```
pascal Media NewTrackMedia (Track theTrack, OSType mediaType,
                            TimeScale timeScale, Handle dataRef,
                            OSType dataRefType);
```

`theTrack` Specifies the track for this operation. Your application obtains this track identifier from such Movie Toolbox functions as `NewMovieTrack` (described on page 2-151).

`mediaType` Specifies the type of media to create. The Movie Toolbox uses this value to find the correct media handler for the new media. If the toolbox cannot locate an appropriate media handler, it returns an error. The following types are available:

`VideoMediaType`	Video media
`SoundMediaType`	Sound media
`TextMediaType`	Text media

`timeScale` Defines the media's time coordinate system.

`dataRef` Specifies the data reference. This parameter contains a handle to the information that identifies the file that contains this media's data. The type of information stored in that handle depends upon the value of the `dataRefType` parameter.

If you are creating a new media that refers to existing media data, you can use the `GetMediaDataRef` function (described on page 2-217) to obtain information about the existing data reference. You can then supply information about that reference to this function.

Set this parameter to `nil` to use the file that is associated with the movie or if the movie does not have a movie file. For example, if you have created the movie using the `CreateMovieFile` function (described on page 2-96) or the `NewMovieFromFile` function (described on page 2-88), the Movie Toolbox assumes that the movie's data resides in the file specified at that time. If you have created the movie using the `NewMovieFromScrap` or `NewMovie` functions (described on page 2-245 and page 2-92, respectively), the movie does not have a movie file.

`dataRefType` Specifies the type of data reference. If the data reference is an alias, you must set this parameter to `rAliasType` (`'alis'`), indicating that the reference is an alias. See *Inside Macintosh: Files* for more information about aliases and the Alias Manager.

If you are creating a new media that refers to existing media data, you can use the `GetMediaDataRef` function (described on page 2-217) to obtain information about the existing data reference. You can then supply information about that reference to this function.

Set this parameter to `nil` to use the file that is associated with the movie or if the movie does not have a movie file. For example, if you have created the movie using the `CreateMovieFile` function (described on page 2-96) or the `NewMovieFromFile` function (described on page 2-88), the Movie Toolbox assumes that the movie's data resides in the file specified at that time. If you have created the movie using the `NewMovieFromScrap` or `NewMovie` functions (described on page 2-245 and page 2-92, respectively), the movie does not have a movie file.

DESCRIPTION

The `NewTrackMedia` function returns a media identifier. If the function cannot create the new media, it sets this returned value to `nil`.

ERROR CODES

`cantFindHandler`	–2003	Cannot locate a handler
`cantOpenHandler`	–2004	Cannot open a handler
`noMediaHandler`	–2006	Media has no media handler
`invalidTrack`	–2009	This track is corrupted or invalid
`invalidTime`	–2015	This time value is invalid

Memory Manager errors

DisposeTrackMedia

The `DisposeTrackMedia` function removes a media from a track. This function does not remove the track from its movie.

```
pascal void DisposeTrackMedia (Media theMedia);
```

`theMedia` Specifies the media for this operation. Your application obtains this media identifier from such Movie Toolbox functions as `NewTrackMedia` and `GetTrackMedia` (described on page 2-153 and page 2-206, respectively).

SPECIAL CONSIDERATIONS

Your application should not call the `DisposeTrackMedia` function as part of the process of disposing of a movie. When you dispose of a movie by calling `DisposeMovie`, the Movie Toolbox disposes of all the movie's tracks and their associated media structures.

ERROR CODES

`invalidMedia`	–2008	This media is corrupted or invalid

Working With Progress and Cover Functions

The Movie Toolbox allows your application to assign two types of custom functions: progress functions and cover functions. These functions allow you to perform special processing under certain circumstances.

Some Movie Toolbox functions can take a long time to execute. For example, if you call the `FlattenMovie` function and specify a large movie, the Movie Toolbox must read and write all the sample data for the movie. During such operations you may wish to display some kind of progress indicator to the user.

A progress function is an application-defined function that you can use to track the progress of time-consuming activities, and thereby keep the user informed about that progress.

The Movie Toolbox allows your application to perform custom processing whenever one of your movie's tracks covers a screen region or reveals a region that was previously covered. You perform this processing in 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 that was previously covered. 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.

Note
The Movie Toolbox does not call your cover function in response to changes to the movie's transformation matrix (for example, changing the matrix by calling the `SetMovieBox` function, which is described on page 2-161, does not cause your cover function to be invoked). u

For a complete discussion of progress and cover functions, see "Application-Defined Functions," which begins on page 2-354.

The `SetMovieProgressProc` function helps your application work with progress functions and the `SetMovieCoverProcs` function helps your application work with cover functions.

SetMovieProgressProc

The `SetMovieProgressProc` function allows you to attach a progress function to each movie. The function will be called whenever a long operation is underway. The Movie Toolbox indicates the progress of the operation to your progress function.

The Movie Toolbox ensures that your progress function is called regularly, but not too often. In addition, the toolbox calls your function only during long operations.

```
pascal void SetMovieProgressProc (Movie theMovie,
                                  MovieProgressProcPtr p,
                                  long refCon);
```

`theMovie` Specifies the movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

`p` Points to your progress function. To remove a movie's progress function, set this parameter to `nil`. Set this parameter to –1 for the Movie Toolbox to provide a default progress function. See "Progress Functions" beginning on page 2-354 for the interface your progress function must support.

`refCon` Specifies a reference constant. The Movie Toolbox passes this value to your progress function.

DESCRIPTION

The following Movie Toolbox functions use progress functions: `ConvertFileToMovieFile` (described on page 2-93), `CutMovieSelection` (described on page 2-247), `CopyMovieSelection` (described on page 2-248), `AddMovieSelection` (described on page 2-250), and `InsertMovieSegment` (described on page 2-257).

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid

SetMovieCoverProcs

The `SetMovieCoverProcs` function allows you to set both types of cover functions.

```
pascal void SetMovieCoverProcs (Movie theMovie,
                                MovieRgnCoverProc uncoverProc,
                                MovieRgnCoverProc coverProc,
                                long refcon);
```

`theMovie` Specifies the movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

`uncoverProc`
: Points to a cover function. This function is called whenever one of your movie's tracks is removed from the screen or resized, revealing a previously hidden screen region. If you want to remove the cover function, set this parameter to `nil`. When the `uncoverProc` parameter is `nil`, `SetMovieCoverProcs` uses the default cover or uncover function. The default cover function does nothing. The default uncover function erases the uncovered area. See "Cover Functions" beginning on page 2-357 for the interface your cover function must support.

`coverProc`
: Points to a cover function. The Movie Toolbox calls this function whenever one of your movies covers a portion of the screen. If you want to remove the cover function, set this parameter to `nil`. See "Cover Functions" beginning on page 2-357 for the interface your cover function must support.

`refcon`
: Specifies a reference constant. The Movie Toolbox passes this value to your cover functions.

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid

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 Movie Spatial Characteristics" describes a number of functions that allow you to work with the display characteristics of movies
- "Working With Sound Volume" discusses the functions that your application can use to work with the sound volume of a movie or a track
- "Working with Movie Time" discusses several functions that allow your application to change the time characteristics of movies
- "Working With Track Time" describes functions that your application can use to change the time characteristics of individual tracks within a movie
- "Working With Media Time" discusses the functions that your application can use to change the time characteristics of a media
- "Finding Interesting Times" describes the Movie Toolbox functions that allow you to retrieve information about when key events occur in movies, tracks, and media structures
- "Locating a Movie's Tracks and Media Structures" describes the functions that allow your application to find tracks that are associated with a movie
- "Working With Alternate Tracks" discusses the Movie Toolbox functions that allow you to define and use alternate tracks in a movie

- "Working With Data References" describes the Movie Toolbox functions that allow you to work with a movie's data references
- "Determining Movie Creation and Modification Time"discusses the functions that you can use to determine when a movie was created or last changed
- "Working With Media Samples" describes several functions that allow you to get and set detailed information about sample data in a media
- "Working With Movie User Data" discusses the functions that you can use to get and set the user data that is associated with a movie

Working With Movie Spatial 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. See the discussion of spatial properties in "About Movies" on page 2-14.

You can use the `SetMovieGWorld` and `GetMovieGWorld` functions to work with a movie's graphics world. See *Inside Macintosh: Imaging* for more information about graphics worlds.

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 described in "Matrix Functions" beginning on page 2-341.

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. u

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.

SetMovieGWorld

The SetMovieGWorld function allows your application to establish a movie's display coordinate system by setting the graphics world for displaying a movie.

```
pascal void SetMovieGWorld (Movie theMovie, CGrafPtr port,
                            GDHandle gdh);
```

theMovie	Specifies the movie for this operation. Your application obtains this movie identifier from such functions as NewMovie, NewMovieFromFile, and NewMovieFromHandle (described on page 2-92, page 2-88, and page 2-90, respectively).
port	Points to the movie's graphics port structure or graphics world. Set this parameter to nil to use the current graphics port.
gdh	Contains a handle to the movie's graphics device structure. Set this parameter to nil to use the current device. If the port parameter specifies a graphics world, set this parameter to nil to use that graphics world's graphics device.

DESCRIPTION

The default cover function provided by the Movie Toolbox uses the background color and pattern from the movie's graphics world during erase operations.

SPECIAL CONSIDERATIONS

The Movie Toolbox automatically sets the graphics world when you create a new movie. Be sure that your application's graphics port is valid or that you specify a valid graphics port with the `port` parameter. If you pass `nil` for the `port` parameter, make sure the current graphics world is valid.

When you use `SetMovieGWorld`, the Movie Toolbox remembers the current background color and background pattern. These are used for erasing in the default movie uncover function.

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid

SEE ALSO

You can retrieve a movie's graphics world by calling the `GetMovieGWorld` function, which is described in the next section.

GetMovieGWorld

Your application can determine a movie's graphics world by calling the `GetMovieGWorld` function.

```
pascal void GetMovieGWorld (Movie theMovie, CGrafPtr *port,
                            GDHandle *gdh);
```

`theMovie` Specifies the movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

`port` Contains a pointer to a field that is to receive a pointer to a graphics port structure. The Movie Toolbox returns a pointer to the movie's graphics port structure. Set this parameter to `nil` if you do not want this information.

`gdh` Contains a pointer to a field that is to receive a handle to a graphics device structure. The Movie Toolbox returns a handle to the movie's graphics device structure. Set this parameter to `nil` if you do not want this information.

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid

SEE ALSO

You can set a movie's graphics world by calling the `SetMovieGWorld` function, which is described in the previous section.

SetMovieBox

The `SetMovieBox` function sets a movie's boundary rectangle, or movie box, which is a rectangle that encompasses the spatial representation of all of the movie's enabled tracks. The movie box is in the display coordinate system.

```
pascal void SetMovieBox (Movie theMovie, const Rect *boxRect);
```

`theMovie`	Specifies the movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).
`boxRect`	Contains a pointer to a rectangle that contains the coordinates of the new boundary rectangle.

DESCRIPTION

The Movie Toolbox changes the rectangle by modifying the translation and scale values of the movie's matrix to accommodate the new boundary rectangle.

The movie box might not have its upper-left corner set at (0,0) in its display window when the movie is first loaded. Consequently, your application may need to adjust the position of the movie box so that it appears in the appropriate location within your application's document window. If you don't reset the movie position, the movie might not be visible when it starts playing.

The following sample code demonstrates how to move the boundary rectangle.

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

SPECIAL CONSIDERATIONS

The `SetMovieBox` function does not call your cover functions.

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid

Memory Manager errors

SEE ALSO

You can modify the movie's matrix directly by calling the `SetMovieMatrix` function, which is described on page 2-170. You can retrieve a movie's boundary rectangle by calling the `GetMovieBox` function, which is described in the next section.

GetMovieBox

The `GetMovieBox` function returns a movie's boundary rectangle, which is a rectangle that encompasses all of the movie's enabled tracks. The movie box is in the coordinate system of the movie's graphics world and defines the movie's boundaries over the entire duration of the movie. The movie's boundary rectangle defines the size and shape of the movie before the Movie Toolbox applies the display clipping region.

```
pascal void GetMovieBox (Movie theMovie, Rect *boxRect);
```

`theMovie` Specifies the movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

`boxRect` Contains a pointer to a rectangle. The `GetMovieBox` function returns the coordinates of the movie's boundary rectangle into the structure referred to by this parameter.

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid

Memory Manager errors

SEE ALSO

You can use the `SetMovieBox` function, which is described in the previous section, to change the coordinates of a movie's boundary rectangle.

GetMovieDisplayBoundsRgn

The `GetMovieDisplayBoundsRgn` function allows your application to determine a movie's display boundary region. The display boundary region encloses all of a movie's enabled tracks after the track matrix, track clip, movie matrix, and movie clip have been applied to all of the movie's tracks. This region is in the display coordinate system of the movie's graphics world. The movie's boundary rectangle encloses this region. For more on boundary regions and matrices for movies and tracks, see "Spatial Properties," which begins on page 2-20.

```
pascal RgnHandle GetMovieDisplayBoundsRgn (Movie theMovie);
```

`theMovie` Specifies the movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

DESCRIPTION

The Movie Toolbox derives the display boundary region only from enabled tracks, and only from those tracks that are used in the current display mode (that is, movie, poster, or preview). The display boundary region is valid for the current movie time.

The `GetMovieDisplayBoundsRgn` function allocates the region and returns a handle to the region. Your application must dispose of this handle when you are done with it. If the movie does not have a spatial representation at the current movie time, the function returns an empty region. If the function could not satisfy your request, it sets the returned handle to `nil`.

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid

Memory Manager errors

SEE ALSO

If you want to determine the boundary region that applies to a time segment of a movie, you can use the `GetMovieSegmentDisplayBoundsRegion` function, which is described in the next section.

GetMovieSegmentDisplayBoundsRgn

The `GetMovieSegmentDisplayBoundsRgn` function allows your application to determine a movie's display boundary region during a specified segment. The display boundary region encloses all of a movie's enabled tracks after the track matrix, track clip, movie matrix, and movie clip have been applied to all of the movie's tracks. This region is in the display coordinate system. The movie's boundary encloses this region. For more on boundary regions and matrices for movies and tracks, see "Spatial Properties," which begins on page 2-20.

```
pascal RgnHandle GetMovieSegmentDisplayBoundsRgn (Movie theMovie,
                                                  TimeValue time,
                                                  TimeValue
                                                  duration);
```

`theMovie` Specifies the movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

`time` Specifies the starting time of the movie segment to consider. This time value must be expressed in the movie's time coordinate system. The `duration` parameter specifies the length of the segment.

`duration` Specifies the length of the segment to consider. Set this parameter to 0 to specify an instant in time.

DESCRIPTION

The Movie Toolbox derives the display boundary region only from enabled tracks and only from those tracks that are used in the current display mode (that is, movie, poster, or preview). If you want to determine the boundary region that applies to the current movie time, you can use `GetMovieDisplayBoundsRegion`, which is described in the previous section.

The `GetMovieSegmentDisplayBoundsRgn` function allocates the region and returns a handle to the region. Your application must dispose of this region when you are done with it. If the movie does not have a spatial representation during the specified segment, the function returns an empty region. If the function could not satisfy your request, it sets the returned handle to `nil`.

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid
`invalidDuration`	–2014	This duration value is invalid
`invalidTime`	–2015	This time value is invalid

Memory Manager errors

SetMovieDisplayClipRgn

The `SetMovieDisplayClipRgn` function allows your application to establish a movie's current display clipping region.

```
pascal void SetMovieDisplayClipRgn (Movie theMovie,
                                    RgnHandle theClip);
```

`theMovie` Specifies the movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

`theClip` Contains a handle to the movie's display clipping region. Note that the Movie Toolbox makes a copy of this region. Your application must dispose of the region referred to by this parameter when you are done with it. Set this parameter to `nil` to disable a movie's clipping region.

DESCRIPTION

The display clipping region defines any final clipping that is applied to the movie before it is displayed, and it is valid for the entire duration of the movie. You must use this region to clip a movie because the Movie Toolbox ignores the clip region of the movie's graphics world during display processing.

Note that the display clipping region is not saved with the movie.

SPECIAL CONSIDERATIONS

Do not use the `SetMovieDisplayClipRgn` function when you are using a movie controller component—use the movie controller component function `MCSetClip` instead. For details on the `MCSetClip` function, see the chapter "Movie Controller Components" in *Inside Macintosh: QuickTime Components*.

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid

Memory Manager errors

SEE ALSO

You can retrieve the display clipping region by calling the `GetMovieDisplayClipRgn` function, which is described in the next section.

GetMovieDisplayClipRgn

The `GetMovieDisplayClipRgn` function allows your application to determine a movie's current display clipping region.

```
pascal RgnHandle GetMovieDisplayClipRgn (Movie theMovie);
```

`theMovie` Specifies the movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

DESCRIPTION

The display clipping region defines the final clipping that is applied to the movie before it is displayed. The display clipping region is valid for the entire duration of the movie.

Note that the display clipping region is not saved with the movie.

The `GetMovieDisplayClipRgn` function allocates the region and returns a handle to the region. Your application must dispose of this region when you are done with it. If the function could not satisfy your request or if there is no display clipping region defined for the movie, the function sets the returned handle to `nil`.

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid

Memory Manager errors

SEE ALSO

You can set the display clipping region by calling the `SetMovieDisplayClipRgn` function, which is described in the previous section.

GetTrackDisplayBoundsRgn

The `GetTrackDisplayBoundsRgn` function allows your application to determine the region a track occupies in a movie's graphics world.

```
pascal RgnHandle GetTrackDisplayBoundsRgn (Track theTrack);
```

`theTrack` Specifies the track for this operation. Your application obtains this track identifier from such Movie Toolbox functions as `NewMovieTrack` and `GetMovieTrack` (described on page 2-151 and page 2-204, respectively).

DESCRIPTION

This region is in the display coordinate system. This region, when intersected with the movie's display clipping region, describes which pixels in the movie's graphics world display information from the specified track. This region is valid for the current movie time.

The `GetTrackDisplayBoundsRgn` function allocates the region and returns a handle to the region. Your application must dispose of this region when you are done with it. If the track does not have a spatial representation at the current movie time, the function returns an empty region. If the function could not satisfy your request, it sets the returned handle to `nil`.

ERROR CODES

`invalidTrack`	–2009	This track is corrupted or invalid

Memory Manager errors

SEE ALSO

If you want to determine the track's boundary region over a specified time segment, you can use the `GetTrackSegmentDisplayBoundsRgn` function, which is described in the next section.

GetTrackSegmentDisplayBoundsRgn

The `GetTrackSegmentDisplayBoundsRgn` function allows your application to determine the region a track occupies in a movie's graphics world during a specified segment.

```
pascal RgnHandle GetTrackSegmentDisplayBoundsRgn (Track theTrack,
                                                  TimeValue time,
                                                  TimeValue duration);
```

`theTrack`	Specifies the track for this operation. Your application obtains this track identifier from such Movie Toolbox functions as `NewMovieTrack` and `GetMovieTrack` (described on page 2-151 and page 2-204, respectively).
`time`	Specifies the starting time of the track segment to consider. This time value must be expressed in the movie's time coordinate system. The `duration` parameter specifies the length of the segment.
`duration`	Specifies the length of the segment to consider. Set this parameter to 0 to consider an instant in time.

DESCRIPTION

This region is in the display coordinate system. When combined with the movie's display clipping region, this region describes which pixels in the movie's graphics world display information from the specified track.

This region is valid for the specified segment.

The `GetTrackSegmentDisplayBoundsRgn` function allocates the region and returns a handle to the region. Your application must dispose of this region when you are done with it. If the track does not have a spatial representation during the specified segment, the function returns an empty region. If the function could not satisfy your request, it sets the returned handle to `nil`.

ERROR CODES

`invalidTrack`	–2009	This track is corrupted or invalid
`invalidDuration`	–2014	This duration value is invalid
`invalidTime`	–2015	This time value is invalid

Memory Manager errors

SEE ALSO

If you want to determine the track's boundary region for the current movie time, you can use the `GetTrackDisplayBoundsRgn` function, which is described in the previous section.

SetTrackLayer

The `SetTrackLayer` function allows your application to set a track's layer.

```
pascal void SetTrackLayer (Track theTrack, short layer);
```

`theTrack` Specifies the track for this operation. Your application obtains this track identifier from such Movie Toolbox functions as `NewMovieTrack` and `GetMovieTrack` (described on page 2-151 and page 2-204, respectively).

`layer` Specifies the track's layer number. Layers are numbered from –32,768 through 32,767. When you create a new track, the Movie Toolbox sets its track number to 0.

DESCRIPTION

Track layers are numbered from –32,768 through 32,767. You can use layers to control how tracks are combined to create a movie. The Movie Toolbox displays layers by layer number. That is, the Movie Toolbox displays higher-numbered layers first, placing lower-numbered layers on top of them. If your movie has more than one track in the

same layer, the Movie Toolbox displays those layers in order by track index value, displaying higher-numbered tracks first.

ERROR CODES

`invalidTrack`	–2009	This track is corrupted or invalid

SEE ALSO

You can retrieve a track's layer number by calling the `GetTrackLayer` function, which is described in the next section.

GetTrackLayer

The `GetTrackLayer` function allows your application to retrieve a track's layer.

```
pascal short GetTrackLayer (Track theTrack);
```

`theTrack` Specifies the track for this operation. Your application obtains this track identifier from such Movie Toolbox functions as `NewMovieTrack` and `GetMovieTrack` (described on page 2-151 and page 2-204, respectively).

DESCRIPTION

The `GetTrackLayer` function returns an integer that contains the track's layer number. Tracks are numbered from –32,768 through 32,767. You can use layers to control how tracks are combined to create a movie. The Movie Toolbox displays layers by layer number. That is, the Movie Toolbox displays higher-numbered layers first, placing lower-numbered layers on top of them. If your movie has more than one track in the same layer, the Movie Toolbox displays those layers in order by track index value, displaying higher-numbered tracks first.

ERROR CODES

`invalidTrack`	–2009	This track is corrupted or invalid

SEE ALSO

You can set a track's layer number by calling the `SetTrackLayer` function, which is described in the previous section.

SetMovieMatrix

The `SetMovieMatrix` function allows your application to set a movie's transformation matrix. The Movie Toolbox uses a movie's matrix to map a movie from its display coordinate system to its graphics world. You can retrieve a movie's matrix with the `GetMovieMatrix` function, which is described in the next section.

```
pascal void SetMovieMatrix (Movie theMovie,
                            const MatrixRecord *matrix);
```

`theMovie` Specifies the movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

`matrix` Contains a pointer to the matrix structure for the movie. If you set this parameter to `nil`, the Movie Toolbox uses the identity matrix.

SPECIAL CONSIDERATIONS

The `SetMovieMatrix` function does not call your cover functions.

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid

SEE ALSO

The Movie Toolbox provides a number of functions that allow you to manipulate movie matrices. See "Matrix Functions," which begins on page 2-341, for information about these functions.

GetMovieMatrix

The `GetMovieMatrix` function allows your application to retrieve a movie's transformation matrix.

```
pascal void GetMovieMatrix (Movie theMovie, MatrixRecord *matrix);
```

`theMovie` Specifies the movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

`matrix` Contains a pointer to a matrix structure. The `GetMovieMatrix` function returns the movie's matrix into the structure referred to by this parameter.

DESCRIPTION

The Movie Toolbox uses a movie's matrix to map a movie from its coordinate system to the display coordinate system.

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid

SEE ALSO

You can set a movie's matrix with the `SetMovieMatrix` function, which is described in the previous section.

The Movie Toolbox provides a number of functions that allow you to manipulate movie matrices. See "Matrix Functions," which begins on page 2-341, for information about these functions.

GetMovieBoundsRgn

The `GetMovieBoundsRgn` function allows your application to determine a movie's boundary region.

```
pascal RgnHandle GetMovieBoundsRgn (Movie theMovie);
```

`theMovie` Specifies the movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

DESCRIPTION

The movie boundary region encloses all of a movie's tracks after the union of the track clip and the track matrix has been applied to all the movie's tracks (but not to the movie itself). This region is in the movie's display coordinate system.

The Movie Toolbox derives the boundary region only from enabled tracks, and only from those tracks that are used in the current display mode (that is, movie or preview). The boundary region is valid for the current movie time.

The `GetMovieBoundsRgn` function allocates the region and returns a handle to the region. Your application must dispose of this region when you are done with it. If the movie does not have a spatial representation at the current time, the function returns an empty region. If the function could not satisfy your request, it sets the returned handle to `nil`.

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid

Memory Manager errors

GetTrackMovieBoundsRgn

The `GetTrackMovieBoundsRgn` function allows your application to determine the region the track occupies in a movie's boundary region. This region is in the display coordinate system of the movie. The Movie Toolbox determines this region by applying the track's clipping region and matrix. This region is valid only for the current movie time.

```
pascal RgnHandle GetTrackMovieBoundsRgn (Track theTrack);
```

`theTrack` Specifies the track for this operation. Your application obtains this track identifier from such Movie Toolbox functions as `NewMovieTrack` and `GetMovieTrack` (described on page 2-151 and page 2-204, respectively).

DESCRIPTION

The `GetTrackMovieBoundsRgn` function allocates the region and returns a handle to the region. Your application must dispose of this region when you are done with it. If the track does not have a spatial representation at the current movie time, the function returns an empty region. If the function could not satisfy your request, it sets the returned handle to `nil`.

ERROR CODES

`invalidTrack`	–2009	This track is corrupted or invalid

SetMovieClipRgn

The `SetMovieClipRgn` function allows your application to establish a movie's clipping region.

```
pascal void SetMovieClipRgn (Movie theMovie, RgnHandle theClip);
```

`theMovie` Specifies the movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

theClip Contains a handle to the movie's clipping region. Note that the Movie Toolbox makes a copy of this region. Your application must dispose of the region referred to by this parameter when you are done with it. Set this parameter to nil to disable clipping for the movie.

DESCRIPTION

The clipping region defines any clipping that is applied to the movie before it is mapped to its graphics world by applying the movie's matrix. The clipping region is in the movie's display coordinate system.

The clipping region is saved with the movie.

ERROR CODES

invalidMovie	–2010	This movie is corrupted or invalid

Memory Manager errors

SEE ALSO

You can retrieve the clipping region by calling the GetMovieClipRgn function, which is described in the next section.

GetMovieClipRgn

The GetMovieClipRgn function allows your application to determine a movie's clipping region.

```
pascal RgnHandle GetMovieClipRgn (Movie theMovie);
```

theMovie Specifies the movie for this operation. Your application obtains this movie identifier from such functions as NewMovie, NewMovieFromFile, and NewMovieFromHandle (described on page 2-92, page 2-88, and page 2-90, respectively).

DESCRIPTION

The clipping region defines any clipping that is applied to the movie before it is mapped to its graphics world by applying the movie's matrix. The clipping region is in the movie's display coordinate system and is valid for the entire duration of the movie.

The GetMovieClipRgn function allocates the region and returns a handle to the region. Your application must dispose of this region when you are done with it. If the function could not satisfy your request or if there is no clipping region defined for the movie, it sets the returned handle to nil.

The clipping region is saved with the movie when your application saves the movie.

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid

Memory Manager errors

SEE ALSO

You can set the clipping region by calling the `SetMovieClipRgn` function, which is described in the previous section.

SetTrackMatrix

The `SetTrackMatrix` function allows your application to establish a track's transformation matrix.

```
pascal void SetTrackMatrix (Track theTrack,
                            const MatrixRecord *matrix);
```

`theTrack` Specifies the track for this operation. Your application obtains this track identifier from such Movie Toolbox functions as `NewMovieTrack` and `GetMovieTrack` (described on page 2-151 and page 2-204, respectively).

`matrix` Contains a pointer to a matrix structure that contains the track's new matrix. If you set this parameter to `nil`, the Movie Toolbox uses the identity matrix.

DESCRIPTION

The Movie Toolbox uses a track's matrix to map a track from its own coordinate system into a movie's display coordinate system.

ERROR CODES

`invalidTrack`	–2009	This track is corrupted or invalid

SEE ALSO

You can get a track's matrix with the `GetTrackMatrix` function, which is described in the next section.

The Movie Toolbox provides a number of functions that allow you to manipulate track matrices. See "Matrix Functions" beginning on page 2-341 for information about these functions.

GetTrackMatrix

The `GetTrackMatrix` function allows your application to retrieve a track's transformation matrix.

```
pascal void GetTrackMatrix (Track theTrack, MatrixRecord *matrix);
```

`theTrack` Specifies the track for this operation. Your application obtains this track identifier from such Movie Toolbox functions as `NewMovieTrack` and `GetMovieTrack` (described on page 2-151 and page 2-204, respectively).

`matrix` Contains a pointer to a matrix structure. The `GetTrackMatrix` function returns the track's matrix into the structure referred to by this parameter.

DESCRIPTION

The Movie Toolbox uses a track's matrix to map a track from its own coordinate system into a movie's display coordinate system.

ERROR CODES

`invalidTrack`	–2009	This track is corrupted or invalid

SEE ALSO

You can set a track's matrix with the `SetTrackMatrix` function, which is described in the previous section.

The Movie Toolbox provides a number of functions that allow you to manipulate track matrices. See "Matrix Functions" on page 2-341 for information about these functions.

GetTrackBoundsRgn

The `GetTrackBoundsRgn` function allows the media to limit the size of the track boundary rectangle. Therefore, the region returned by `GetTrackBoundsRgn` may not be rectangular and may be smaller than the track boundary region.

```
pascal RgnHandle GetTrackBoundsRgn (Track theTrack);
```

`theTrack` Specifies the track for this operation. Your application obtains this track identifier from such Movie Toolbox functions as `NewMovieTrack` and `GetMovieTrack` (described on page 2-151 and page 2-204, respectively).

DESCRIPTION

The `GetTrackBoundsRgn` function allocates the region and returns a handle to the region. Your application must dispose of this region when you are done with it. If the track does not have a spatial representation during the specified segment, the function returns an empty region. If the function could not satisfy your request, it sets the returned handle to `nil`.

ERROR CODES

`invalidTrack`	–2009	This track is corrupted or invalid

Memory Manager errors

SEE ALSO

See the description of the base media handler component's `MediaGetSrcRgn` function in *Inside Macintosh: QuickTime Components* for details on how the media limits the size of the track boundary region.

SetTrackDimensions

The `SetTrackDimensions` function allows your application to establish a track's source, or display, rectangle.

```
pascal void SetTrackDimensions (Track theTrack, Fixed width,
                                Fixed height);
```

`theTrack`	Specifies the track for this operation. Your application obtains this track identifier from such Movie Toolbox functions as `NewMovieTrack` and `GetMovieTrack` (described on page 2-151 and page 2-204, respectively).
`width`	Contains a fixed-point number that specifies the width, in pixels, of the track's rectangle. This value corresponds to the x coordinate of the lower-right corner of the track's rectangle.
`height`	Contains a fixed-point number that specifies the height, in pixels, of the track's rectangle. This value corresponds to the y coordinate of the lower-right corner of the track's rectangle.

DESCRIPTION

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.

If you change the dimensions of an existing track, the media data is scaled to fit into the new rectangle.

ERROR CODES

`invalidTrack`	–2009	This track is corrupted or invalid

SEE ALSO

You can use the `GetTrackDimensions` function, which is described in the next section, to retrieve a track's rectangle.

GetTrackDimensions

The `GetTrackDimensions` function allows your application to determine a track's source, or display, rectangle.

```
pascal void GetTrackDimensions (Track theTrack, Fixed *width,
                                Fixed *height);
```

`theTrack` Specifies the track for this operation. Your application obtains this track identifier from such Movie Toolbox functions as `NewMovieTrack` and `GetMovieTrack` (described on page 2-151 and page 2-204, respectively).

`width` Contains a pointer to a fixed-point number. The Movie Toolbox returns the width, in pixels, of the track's rectangle. This value corresponds to the x coordinate of the lower-right corner of the track's rectangle.

`height` Contains a pointer to a fixed-point number. The Movie Toolbox returns the height, in pixels, of the track's rectangle. This value corresponds to the y coordinate of the lower-right corner of the track's rectangle.

DESCRIPTION

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.

ERROR CODES

`invalidTrack`	–2009	This track is corrupted or invalid

SEE ALSO

You can use the `SetTrackDimensions` function, which is described in the previous section, to set a track's rectangle.

SetTrackClipRgn

The `SetTrackClipRgn` function allows your application to set the clipping region of a track.

```
pascal void SetTrackClipRgn (Track theTrack, RgnHandle theClip);
```

`theTrack` Specifies the track for this operation. Your application obtains this track identifier from such Movie Toolbox functions as `NewMovieTrack` and `GetMovieTrack` (described on page 2-151 and page 2-204, respectively).

`theClip` Contains a handle to the track's clipping region. Note that the Movie Toolbox makes a copy of this region. Your application must dispose of the region referred to by this parameter when you are done with it. Set this parameter to `nil` to disable clipping for the track.

DESCRIPTION

The clipping region is in the track's coordinate system. The Movie Toolbox applies the clipping region to a track before it applies the track's matrix.

ERROR CODES

`invalidTrack`	–2009	This track is corrupted or invalid

Memory Manager errors

SEE ALSO

You can get a track's clipping region by calling the `GetTrackClipRgn` function, which is described in the next section.

GetTrackClipRgn

The `GetTrackClipRgn` function allows your application to determine the clipping region of a track.

```
pascal RgnHandle GetTrackClipRgn (Track theTrack);
```

`theTrack` Specifies the track for this operation. Your application obtains this track identifier from such Movie Toolbox functions as `NewMovieTrack` and `GetMovieTrack` (described on page 2-151 and page 2-204, respectively).

DESCRIPTION

The clipping region is in the track's coordinate system. The Movie Toolbox applies the clipping region to a track before it applies the track's matrix. This region is valid for the entire duration of the track.

The `GetTrackClipRgn` function allocates the region and returns a handle to the region. Your application must dispose of this region when you are done with it. If the function could not satisfy your request or if there is no clipping region defined for the track, it sets the returned handle to `nil`.

ERROR CODES

`invalidTrack`	–2009	This track is corrupted or invalid

Memory Manager errors

SEE ALSO

You can establish a track's clipping region by calling the `SetTrackClipRgn` function, which is described in the previous section.

SetTrackMatte

The `SetTrackMatte` function allows your application to set a track's matte. The matte defines which of the track's pixels are displayed in a movie. You must specify the matte in a pixel map structure.

```
pascal void SetTrackMatte (Track theTrack, PixMapHandle theMatte);
```

`theTrack` Specifies the track for this operation. Your application obtains this track identifier from such Movie Toolbox functions as `NewMovieTrack` and `GetMovieTrack` (described on page 2-151 and page 2-204, respectively).

`theMatte` Contains a handle to the matte. The Movie Toolbox makes a copy of the matte, including its color table and pixels. Consequently, your application must dispose of the matte when you are done with it. Set this parameter to `nil` to remove the track's matte.

DESCRIPTION

The Movie Toolbox displays the weighted average of the track and its destination based on the corresponding pixel in the matte (this feature is fully functional in System 7 and is approximated in System 6).

SPECIAL CONSIDERATIONS

Note that the track matte must have its boundaries defined by the track rectangle.

ERROR CODES

`invalidTrack`	–2009	This track is corrupted or invalid

Memory Manager errors

SEE ALSO

You can retrieve a track's matte by calling the `GetTrackMatte` function, which is described in the next section. Listing 2-15 on page 2-73 shows how to use the `SetTrackMatte` and `GetTrackMatte` functions to create a track matte.

GetTrackMatte

The `GetTrackMatte` function allows your application to retrieve a copy of a track's matte. The matte defines which of the track's pixels are displayed in a movie, and it is valid for the entire duration of the movie. This function returns the matte in a pixel map structure. You may use QuickDraw functions to manipulate the returned matte. However, you should use the Movie Toolbox's `DisposeMatte` function (described in the next section) to dispose of the matte when you are finished with it.

```
pascal PixMapHandle GetTrackMatte (Track theTrack);
```

`theTrack` Specifies the track for this operation. Your application obtains this track identifier from such Movie Toolbox functions as `NewMovieTrack` and `GetMovieTrack` (described on page 2-151 and page 2-204, respectively).

DESCRIPTION

The `GetTrackMatte` function returns a handle to the matte. Your application must dispose of this handle when you are done with it—you must use the `DisposeMatte` function, which is described in the next section, to dispose of the matte. If the function could not satisfy your request, it sets the returned handle to `nil`.

ERROR CODES

`invalidTrack`	–2009	This track is corrupted or invalid

Memory Manager errors

SEE ALSO

You can establish a track's matte by calling the `SetTrackMatte` function, which is described in the previous section. Listing 2-15 on page 2-73 shows how to use the `SetTrackMatte` and `GetTrackMatte` functions to create a track matte.

DisposeMatte

The `DisposeMatte` function disposes of a matte that you obtained from the `GetTrackMatte` function, which is described in the previous section.

```
pascal void DisposeMatte (PixMapHandle theMatte);
```

`theMatte` Handle to the matte to be disposed. Your application obtains this handle from the `GetTrackMatte` function.

SPECIAL CONSIDERATIONS

You should not use this function to dispose of mattes or pixel maps that you obtain through other means.

ERROR CODES

None

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.

SetMovieVolume

The `SetMovieVolume` function allows your application to set a movie's current volume.

```
pascal void SetMovieVolume (Movie theMovie, short volume);
```

`theMovie` Specifies the movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

`volume` Specifies the current volume setting of the movie represented as a 16-bit, fixed-point number. The high-order 8 bits contain the integer part of the value; the low-order 8 bits contain the fractional part. Volume values range from –1.0 to 1.0. Negative values play no sound but preserve the absolute value of the volume setting.

`kFullVolume`
Sets the movie to full volume (constant value is 1.0).

`kNoVolume`
Sets the movie to no volume (constant value is 0.0).

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid

SEE ALSO

Your application can obtain the current volume setting by calling the `GetMovieVolume` function, which is described in the next section.

GetMovieVolume

The `GetMovieVolume` function returns a movie's current volume setting.

```
pascal short GetMovieVolume (Movie theMovie);
```

`theMovie` Specifies the movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

DESCRIPTION

The `GetMovieVolume` function returns an integer that contains the movie's current volume represented as a 16-bit, fixed-point number. The high-order 8 bits contain the integer part of the value; the low-order 8 bits contain the fractional part. Volume values

range from –1.0 to 1.0. Negative values play no sound but preserve the absolute value of the volume setting.

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid

SEE ALSO

You can change a movie's current volume by calling the `SetMovieVolume` function, which is described in the previous section.

SetTrackVolume

The `SetTrackVolume` function allows your application to set a track's current volume.

```
pascal void SetTrackVolume (Track theTrack, short volume);
```

`theTrack` Specifies the track for this operation. Your application obtains this track identifier from such Movie Toolbox functions as `NewMovieTrack` and `GetMovieTrack` (described on page 2-151 and page 2-204, respectively).

`volume` Specifies the current volume setting of the track represented as a 16-bit, fixed-point number. The high-order 8 bits contain the integer part of the value; the low-order 8 bits contain the fractional part. Volume values range from –1.0 to 1.0. Negative values play no sound but preserve the absolute value of the volume setting.

`kFullVolume`
Sets the track to full volume (constant value is 1.0).

`kNoVolume` Sets the track to no volume (constant value is 0.0).

DESCRIPTION

Note that, when the track is played, the track's volume is scaled by the volume setting of the movie that contains the track.

ERROR CODES

`invalidTrack`	–2009	This track is corrupted or invalid

SEE ALSO

Your application can obtain the current volume setting by calling the `GetTrackVolume` function, which is described in the next section.

GetTrackVolume

The `GetTrackVolume` function returns a track's current volume setting.

```
pascal short GetTrackVolume (Track theTrack);
```

`theTrack` Specifies the track for this operation. Your application obtains this track identifier from such Movie Toolbox functions as `NewMovieTrack` and `GetMovieTrack` (described on page 2-151 and page 2-204, respectively).

DESCRIPTION

The `GetTrackVolume` function returns an integer that contains the track's current volume represented as a 16-bit, fixed-point number. The high-order 8 bits contain the integer part of the value; the low-order 8 bits contain the fractional part. Volume values range from –1.0 to 1.0. Negative values play no sound but preserve the absolute value of the volume setting.

ERROR CODES

`invalidTrack`	–2009	This track is corrupted or invalid

SEE ALSO

You can change a track's current volume by calling the `SetTrackVolume` function, which is described in the previous section.

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. For a complete discussion of the relationships between movie, track, and media time parameters, see "Introduction to Movies" beginning on page 2-5. For information about more functions that work with time, see "Time Base Functions" beginning on page 2-315.

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.

GetMovieDuration

The `GetMovieDuration` function returns the duration of a movie. The Movie Toolbox examines the durations of all the tracks of the movie to determine this value.

```
pascal TimeValue GetMovieDuration (Movie theMovie);
```

`theMovie` Specifies the movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

DESCRIPTION

The `GetMovieDuration` function returns a time value. This time value indicates the movie's duration, and it is expressed in the movie's time scale.

You cannot set movie direction explicitly because it is calculated as being the maximum durations of all the tracks in the movie.

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid

SetMovieTimeValue

The `SetMovieTimeValue` function allows your application to set a movie's time value. You specify the new time as a time value, rather than in a time structure. You must ensure that the time value is in the movie's time scale.

```
pascal void SetMovieTimeValue (Movie theMovie, TimeValue newTime);
```

`theMovie` Specifies the movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

newTime Specifies the movie's new time value. The Movie Toolbox interprets this time value relative to the movie's time scale. If you specify a value that is outside the duration of the movie, the Movie Toolbox sets the movie time to the beginning or end of the movie, as appropriate.

ERROR CODES

invalidMovie	–2010	This movie is corrupted or invalid

SEE ALSO

You can also set a movie's current time by calling the SetMovieTime function, which is described in the next section. This function requires that you specify the new time value in a time structure.

SetMovieTime

The SetMovieTime function allows your application to change a movie's current time. You must specify the new time in a time structure. The Movie Toolbox saves the movie's current time when you save the movie.

```
pascal void SetMovieTime (Movie theMovie,
                          const TimeRecord *newTime);
```

theMovie Specifies the movie for this operation. Your application obtains this movie identifier from such functions as NewMovie, NewMovieFromFile, and NewMovieFromHandle (described on page 2-92, page 2-88, and page 2-90, respectively).

newTime Contains a pointer to a time structure. If you specify a value that is outside the duration of the movie, the Movie Toolbox sets the movie time to the beginning or end of the movie, as appropriate.

ERROR CODES

invalidMovie	–2010	This movie is corrupted or invalid

SEE ALSO

You can use the SetMovieTimeValue function, described in the previous section, to change a movie's current time without specifying a time structure.

You can retrieve a movie's current time value by calling the GetMovieTime function, which is described in the next section.

GetMovieTime

The `GetMovieTime` function returns a movie's current time. This function returns the time in two formats: as a time value and in a time structure.

```
pascal TimeValue GetMovieTime (Movie theMovie,
                               TimeRecord *currentTime);
```

`theMovie` Specifies the movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

`currentTime` Contains a pointer to a time structure. The `GetMovieTime` function updates this time structure to contain the movie's current time. If you do not want this information, set this parameter to `nil`.

DESCRIPTION

The `GetMovieTime` function returns a time value. This time value indicates the movie's current time, and it is expressed in the movie's time scale.

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid

SEE ALSO

You can set a movie's current time by calling the `SetMovieTime` or `SetMovieTimeValue` functions, which are described on page 2-186 and page 2-185, respectively.

SetMovieRate

The `SetMovieRate` function sets a movie's playback rate.

```
pascal void SetMovieRate (Movie theMovie, Fixed rate);
```

`theMovie` Specifies the movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

`rate` Specifies the new movie rate as a 32-bit, fixed-point number. Positive integers indicate forward rates and negative integers indicate reverse rates.

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid

SEE ALSO

Your application can retrieve a movie's current playback rate by calling the `GetMovieRate` function, which is described in the next section. To play a movie at the movie's preferred rate from a position stored within the movie, you can use the `StartMovie` function (described on page 2-111).

GetMovieRate

The `GetMovieRate` function returns a movie's playback rate.

```
pascal Fixed GetMovieRate (Movie theMovie);
```

`theMovie` Specifies the movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

DESCRIPTION

The `GetMovieRate` function returns the movie rate as a 32-bit, fixed-point number. Positive integers indicate forward rates and negative integers indicate reverse rates.

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid

SEE ALSO

Your application can set the movie's playback rate by calling the `SetMovieRate` function, which is described in the previous section.

SetMovieTimeScale

The `SetMovieTimeScale` function establishes a movie's time scale.

```
pascal void SetMovieTimeScale (Movie theMovie,
                               TimeScale timeScale);
```

`theMovie` Specifies the movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

`timeScale` Specifies the movie's new time scale.

DESCRIPTION

In response to this request, the Movie Toolbox adjusts the edit list of the movie's tracks so that movie playback is unaffected. If you change a movie's time scale by setting it to a smaller value (thereby losing precision in the movie's time values), the Movie Toolbox may edit information from the movie. In general, you should only increase the time scale value, and you should try to use integer multiples of the existing time scale.

SPECIAL CONSIDERATIONS

Do not call `SetMovieTimeScale` if you have edited your movie. This function quantizes the beginning and the end of the edits to the new units. Therefore, if you do not use an integral multiple, the position of your edits may change.

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid

SEE ALSO

You can retrieve a movie's time scale by calling the `GetMovieTimeScale` function, which is described in the next section.

GetMovieTimeScale

The `GetMovieTimeScale` function returns the time scale of a movie.

```
pascal TimeScale GetMovieTimeScale (Movie theMovie);
```

`theMovie` Specifies the movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

DESCRIPTION

The default QuickTime movie time scale is 600 units per second; however, this number may change in the future. The default time scale was chosen because it is convenient for working with common video frame rates of 30, 25, 24, 15, 12, 10, and 8.

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid

SEE ALSO

You can set a movie's time scale by calling the `SetMovieTimeScale` function, which is described in the previous section.

GetMovieTimeBase

The `GetMovieTimeBase` function returns a movie's time base.

```
pascal TimeBase GetMovieTimeBase (Movie theMovie);
```

`theMovie` Specifies the movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

DESCRIPTION

You cannot use the returned time base value with the Movie Toolbox's `SetTimeBaseMasterTimeBase` and `SetTimeBaseMasterClock` functions (described on page 2-320 and page 2-318, respectively). Use the `SetMovieMasterTimeBase` and `SetMovieMasterClock` functions (described on page 2-318 and page 2-317, respectively) instead.

The Movie Toolbox disposes of a movie's time base when you dispose of the movie.

SPECIAL CONSIDERATIONS

Do not dispose of the `TimeBase` result returned by the `GetMovieTimeBase` function as it is owned by the movie.

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid

Working With Track Time

The Movie Toolbox provides several functions that allow your application to determine and establish a track's time parameters. A track uses the time base of the movie that contains the track; therefore there are no functions that work with a track's time base or time scale. However, you can determine a track's duration and its offset from the start of a movie.

All of the tracks in a movie use the movie's time coordinate system. That is, the movie's time scale defines the basic time unit for each of the movie's tracks. Each track begins at the beginning of the movie, but the track's data might not begin until some time value other than 0. This intervening time is represented by blank space—in an audio track the blank space translates to silence; in a video track the blank space generates no visual image. This blank space is the **track offset.** Each track has its own **duration.** This duration need not correspond to the duration of the movie. A movie duration always equals the maximum track duration. See Figure 2-6 on page 2-12 for a visual representation of track duration and track offset.

You can use the `GetTrackDuration` function to determine a track's duration.

The `SetTrackOffset` and `GetTrackOffset` functions enable you to work with a track's offset from the start of the movie that contains it.

The `TrackTimeToMediaTime` function lets you translate a track's time to the corresponding time value of a media in the track.

GetTrackDuration

The `GetTrackDuration` function returns the duration of a track. The duration corresponds to the ending time of the track in the movie's time coordinate system (remember that all tracks start at movie time 0).

```
pascal TimeValue GetTrackDuration (Track theTrack);
```

`theTrack` Specifies the track for this operation. Your application obtains this track identifier from such Movie Toolbox functions as `NewMovieTrack` and `GetMovieTrack` (described on page 2-151 and page 2-204, respectively).

DESCRIPTION

The `GetTrackDuration` function returns a time value. This time value indicates the track's duration, and it is expressed in the time scale of the movie that contains the track.

ERROR CODES

`invalidTrack`	–2009	This track is corrupted or invalid

SetTrackOffset

The `SetTrackOffset` function modifies the duration of the empty space that lies at the beginning of the track, thus changing the duration of the entire track. You specify this time offset as a time value in the movie's time scale. See Figure 2-6 on page 2-12 for an illustration of a track offset in a movie.

```
pascal void SetTrackOffset (Track theTrack,
                            TimeValue movieOffsetTime);
```

`theTrack` Specifies the track for this operation. Your application obtains this track identifier from such Movie Toolbox functions as `NewMovieTrack` and `GetMovieTrack` (described on page 2-151 and page 2-204, respectively).

`movieOffsetTime`
Specifies the track's offset from the start of the movie, and must be expressed in the time scale of the movie that contains the track.

ERROR CODES

`invalidTrack`	–2009	This track is corrupted or invalid

SEE ALSO

You can determine a track's time offset by calling the `GetTrackOffset` function, which is described in the next section.

GetTrackOffset

The `GetTrackOffset` function allows your application to determine the time difference between the start of a track and the start of the movie that contains the track.

```
pascal TimeValue GetTrackOffset (Track theTrack);
```

`theTrack` Specifies the track for this operation. Your application obtains this track identifier from such Movie Toolbox functions as `NewMovieTrack` and `GetMovieTrack` (described on page 2-151 and page 2-204, respectively).

DESCRIPTION

The `GetTrackOffset` function returns a time value. This time value indicates the track's offset from the start of the movie, and it is expressed in the time scale of the movie that contains the track.

ERROR CODES

`invalidTrack`	–2009	This track is corrupted or invalid

SEE ALSO

You can set a track's offset by calling the `SetTrackOffset` function, which is described in the previous section.

TrackTimeToMediaTime

The `TrackTimeToMediaTime` function allows your application to convert a track's time value to a time value that is appropriate to the track's media using the track's edit list. You specify the track's time in the movie's time coordinate system.

```
pascal TimeValue TrackTimeToMediaTime (TimeValue value,
                                       Track theTrack);
```

`value` Specifies the track's time value; must be expressed in the time scale of the movie that contains the track.

`theTrack` Specifies the track for this operation. Your application obtains this track identifier from such Movie Toolbox functions as `NewMovieTrack` and `GetMovieTrack` (described on page 2-151 and page 2-204, respectively).

DESCRIPTION

The Movie Toolbox returns a value that is in the media's time coordinate system.

You can use the `TrackTimeToMediaTime` function to determine whether a specified track edit is empty. If the track time corresponds to empty space, this function returns a value of –1.

The `TrackTimeToMediaTime` function maps the track time through the track's edit list to come up with the media time. This time value contains the track's time value according to the media's time coordinate system. If the time you specified lies outside of the movie's active segment or corresponds to empty space in the track, the `TrackTimeToMediaTime` function returns a value of –1.

ERROR CODES

`invalidTrack`	–2009	This track is corrupted or invalid

Working With Media Time

The Movie Toolbox provides functions that allow your application to work with the time parameters of a media.

You can use the `GetMediaDuration` function to determine a media's duration.

The `GetMediaTimeScale` and `SetMediaTimeScale` let you determine or establish a media's time scale.

GetMediaDuration

The `GetMediaDuration` function returns the duration of a media.

```
pascal TimeValue GetMediaDuration (Media theMedia);
```

`theMedia` Specifies the media for this operation. Your application obtains this media identifier from such Movie Toolbox functions as `NewTrackMedia` and `GetTrackMedia` (described on page 2-153 and page 2-206, respectively).

DESCRIPTION

The `GetMediaDuration` function returns a time value. This time value indicates the media's duration, and it is expressed in the time scale of the media.

ERROR CODES

`invalidMedia`	–2008	This media is corrupted or invalid

SetMediaTimeScale

The `SetMediaTimeScale` function allows your application to set a media's time scale.

```
pascal void SetMediaTimeScale (Media theMedia,
                               TimeScale timeScale);
```

`theMedia` Specifies the media for this operation. Your application obtains this media identifier from such Movie Toolbox functions as `NewTrackMedia` and `GetTrackMedia` (described on page 2-153 and page 2-206, respectively).

`timeScale` Specifies the media's new time scale.

DESCRIPTION

In response to this request, the Movie Toolbox attempts to adjust the edit list of the appropriate track so that movie playback is unaffected. If you change a media's time scale by setting it to a smaller value, you may lose precision in media time values. In general, you should only increase the time scale value, and you should try to use integer multiples of the existing time scale.

SPECIAL CONSIDERATIONS

Do not use `SetMediaTimeScale` as a general rule. If you call this function with a number that is not an integer multiple, the duration of the samples vary unpredictably, and their start times tend to drift.

ERROR CODES

`invalidMedia`	–2008	This media is corrupted or invalid

GetMediaTimeScale

The `GetMediaTimeScale` function allows your application to determine a media's time scale.

```
pascal TimeScale GetMediaTimeScale (Media theMedia);
```

`theMedia` Specifies the media for this operation. Your application obtains this media identifier from such Movie Toolbox functions as `NewTrackMedia` and `GetTrackMedia` (described on page 2-153 and page 2-206, respectively).

DESCRIPTION

The `GetMediaTimeScale` function returns the media's time scale.

ERROR CODES

`invalidMedia`	–2008	This media is corrupted or invalid

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.** For more information on key frames, see the chapter "Image Compression Manager" in this book. 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.

GetMovieNextInterestingTime

The `GetMovieNextInterestingTime` function searches for times of interest in a movie. This function examines only the movie's enabled tracks.

```
pascal void GetMovieNextInterestingTime (Movie theMovie,
                                         short interestingTimeFlags,
                                         short numMediaTypes,
                                         const OSType *whichMediaTypes,
                                         TimeValue time, Fixed rate,
                                         TimeValue *interestingTime,
                                         TimeValue *interestingDuration);
```

`theMovie` Specifies the movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

`interestingTimeFlags`
Specifies the search criteria. Note that you may set only one of the `nextTimeMediaSample`, `nextTimeMediaEdit`, `nextTimeTrackEdit` and `nextTimeSyncSample` flags to 1. The following flags are available (set unused flags to 0):

`nextTimeMediaSample`
Searches for the next sample in the movie's media. Set this flag to 1 to search for the next sample.

`nextTimeMediaEdit`
Searches for the next group of samples in the movie's media. Set this flag to 1 to search for the next group of samples.

`nextTimeTrackEdit`
Searches for the media sample that corresponds to the next entry in a track's media edit list. The end of the track is considered an empty edit. Set this flag to 1 to search for the next track edit.

`nextTimeSyncSample`
Searches for the next sync sample in the movie's media. Set this flag to 1 to search for the next 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.

nextTimeEdgeOK
: Instructs the Movie Toolbox that you are willing to receive information about elements that begin or end at the time specified by the `time` parameter. Set this flag to 1 to accept this information.

  This flag is especially useful at the beginning or end of a movie. The function returns valid information about the beginning and end of the movie.

nextTimeIgnoreActiveSegment
: Instructs the Movie Toolbox to look outside of the active segment for samples that meet the search criteria. Set this flag to 1 to search outside of the active segment.

`numMediaTypes`
: Specifies the number of media types in the table referred to by the `whichMediaType` parameter. Set this parameter to 0 to search all media types.

`whichMediaTypes`
: Contains a pointer to an array of media types. You can use this parameter to limit the search to a specified set of media types. Each entry in the table referred to by this parameter identifies a media type to be included in the search. You use the `numMediaTypes` parameter to indicate the number of entries in the table. Set this parameter to `nil` to search all media types.

  `VisualMediaCharacteristic 'eyes'`
  : Instructs the Movie Toolbox to search all tracks that have spatial bounds.

  `AudioMediaCharacteristic 'ears'`
  : Instructs the Movie Toolbox to search all tracks that play sound.

`time`
: Specifies a time value that establishes the starting point for the search. This time value must be expressed in the movie's time scale.

`rate`
: Contains the search direction. Negative values cause the Movie Toolbox to search backward from the starting point specified in the `time` parameter. Other values cause a forward search.

`interestingTime`
: Contains a pointer to a time value. The Movie Toolbox returns the first time value it finds that meets the search criteria specified in the `flags` parameter. This time value is in the movie's time scale.

  If there are no times that meet the search criteria you specify, the Movie Toolbox sets this value to –1.

  If you are not interested in this information, set this parameter to `nil`.

`interestingDuration`
: Contains a pointer to a time value. The Movie Toolbox returns the duration of the interesting time. This time value is in the movie's time coordinate system. Set this parameter to `nil` if you do not want this information—in this case, the function works more quickly.

DESCRIPTION

You can use the `GetMovieNextInterestingTime` function to step through the frames of a movie one by one. If no tracks match the media types, the `invalidMedia` error is returned.

ERROR CODES

`invalidMedia`	–2008	This media is corrupted or invalid
`invalidTrack`	–2009	This track is corrupted or invalid
`invalidMovie`	–2010	This movie is corrupted or invalid
`invalidTime`	–2015	This time value is invalid

GetTrackNextInterestingTime

The `GetTrackNextInterestingTime` function searches for times of interest in a track.

```
pascal void GetTrackNextInterestingTime (Track theTrack,
                                         short interestingTimeFlags,
                                         TimeValue time, Fixed rate,
                                         TimeValue *interestingTime,
                                         TimeValue *interestingDuration);
```

`theTrack` Specifies the track for this operation. Your application obtains this track identifier from such Movie Toolbox functions as `NewMovieTrack` and `GetMovieTrack` (described on page 2-151 and page 2-204, respectively).

`interestingTimeFlags`
Specifies the search criteria. Note that you may set only one of the `nextTimeMediaSample`, `nextTimeMediaEdit`, `nextTimeTrackEdit` and `nextTimeSyncSample` flags to 1. The following flags are available (set unused flags to 0):

`nextTimeMediaSample`
Searches for the next sample in the track's media. Set this flag to 1 to search for the next sample.

`nextTimeMediaEdit`
Searches for the next group of samples in the track's media. Set this flag to 1 to search for the next group of samples.

`nextTimeTrackEdit`
Searches for the media sample that corresponds to the next entry in a track's media edit list. The end of the track is considered an empty edit. Set this flag to 1 to search for the next track edit.

nextTimeSyncSample
: Searches for the next sync sample in the track's media. Set this flag to 1 to search for the next 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.

nextTimeEdgeOK
: Instructs the Movie Toolbox that you are willing to receive information about elements that begin or end at the time specified by the `time` parameter. Set this flag to 1 to accept this information.

  This flag is especially useful at the beginning or end of a track. The function returns valid information about the beginning and end of the track.

nextTimeIgnoreActiveSegment
: Instructs the Movie Toolbox to look outside of the active segment for samples that meet the search criteria. Set this flag to 1 to search outside of the active segment.

`time` Specifies a time value that establishes the starting point for the search. This time value must be expressed in the movie's time scale.

`rate` Contains the search direction. Negative values cause the Movie Toolbox to search backward from the starting point specified in the `time` parameter. Other values cause a forward search.

`interestingTime`
: Contains a pointer to a time value. The Movie Toolbox returns the first time value it finds that meets the search criteria specified in the `flags` parameter. This time value is in the movie's time scale.

  If there are no times that meet the search criteria you specify, the Movie Toolbox sets this value to –1.

  Set this parameter to `nil` if you are not interested in this information.

`interestingDuration`
: Contains a pointer to a time value. The Movie Toolbox returns the duration of the interesting time. This time value is in the movie's time coordinate system. Set this parameter to `nil` if you do not want this information—in this case, the function works more quickly.

ERROR CODES

`invalidTrack`	–2009	This track is corrupted or invalid
`invalidTime`	–2015	This time value is invalid

GetMediaNextInterestingTime

The `GetMediaNextInterestingTime` function searches for times of interest in a media.

```
pascal void GetMediaNextInterestingTime (Media theMedia,
                                         short interestingTimeFlags,
                                         TimeValue time, Fixed rate,
                                         TimeValue *interestingTime,
                                         TimeValue *interestingDuration);
```

`theMedia` Specifies the media for this operation. Your application obtains this media identifier from such Movie Toolbox functions as `NewTrackMedia` and `GetTrackMedia` (described on page 2-153 and page 2-206, respectively).

`interestingTimeFlags`
Specifies the search criteria. Note that you may set only one of the `nextTimeMediaSample`, `nextTimeMediaEdit` and `nextTimeSyncSample` flags to 1. The following flags are available (set unused flags to 0):

`nextTimeMediaSample`
Searches for the next sample in the media. Set this flag to 1 to search for the next sample.

`nextTimeMediaEdit`
Searches for the next group of samples in the media. Set this flag to 1 to search for the next group of samples.

`nextTimeSyncSample`
Searches for the next sync sample in the media. Set this flag to 1 to search for the next 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.

`nextTimeEdgeOK`
Instructs the Movie Toolbox that you are willing to receive information about elements that begin or end at the time specified by the `time` parameter. Set this flag to 1 to accept this information.

This flag is especially useful at the beginning or end of a media. The function returns valid information about the beginning and end of the media.

`time` Specifies a time value that establishes the starting point for the search. This time value must be expressed in the media's time scale.

`rate` Contains the search direction. Negative values cause the Movie Toolbox to search backward from the starting point specified in the `time` parameter. Other values cause a forward search.

`interestingTime`
Contains a pointer to a time value. The Movie Toolbox returns the first time value it finds that meets the search criteria specified in the `flags` parameter. This time value is in the media's time scale.

If there are no times that meet the search criteria you specify, the Movie Toolbox sets this value to –1.

Set this parameter to `nil` if you are not interested in this information.

`interestingDuration`
Contains a pointer to a time value. The Movie Toolbox returns the duration of the interesting time. This time value is in the media's time coordinate system. Set this parameter to `nil` if you do not want this information—in this case, the function works more quickly.

DESCRIPTION

`GetMediaNextInterestingTime` ignores all the edits that are defined in a movie's tracks.

ERROR CODES

`invalidMedia`	–2008	This media is corrupted or invalid
`invalidTime`	–2015	This time value is invalid

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.

GetMovieTrackCount

The `GetMovieTrackCount` function returns the number of tracks in a movie.

```
pascal long GetMovieTrackCount (Movie theMovie);
```

`theMovie` Specifies the movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid

GetMovieIndTrack

The `GetMovieIndTrack` function allows your application to determine the track identifier of a track given the track's index value. The index value identifies the track among all current tracks in a movie. Index values range from 1 to the number of tracks in the movie.

```
pascal Track GetMovieIndTrack (Movie theMovie, long index);
```

`theMovie` Specifies the movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

`index` Specifies the index value of the track for this operation.

DESCRIPTION

The `GetMovieIndTrack` function returns the track identifier that is appropriate to the specified track. If the function cannot locate the track, it sets this returned value to `nil`.

ERROR CODES

`badTrackIndex`	–2028	This track index value is not valid
`invalidMovie`	–2010	This movie is corrupted or invalid

SEE ALSO

You can determine the number of tracks in a movie by calling the `GetMovieTrackCount` function, which is described in the previous section.

GetMovieTrack

The `GetMovieTrack` function allows your application to determine the track identifier of a track given the track's ID value.

```
pascal Track GetMovieTrack (Movie theMovie, long trackID);
```

`theMovie` Specifies the movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

`trackID` Specifies the ID value of the track for this operation.

DESCRIPTION

The `GetMovieTrack` function returns the track identifier that is appropriate to the specified track. If the function cannot locate the track, it sets this returned value to `nil`.

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid
`trackIDNotFound`	–2029	Cannot locate a track with this ID value

SEE ALSO

You can obtain a track's ID value by calling the `GetTrackID` function, which is described in the next section. You can use a track's index value to obtain its track identifier by calling the `GetMovieIndTrack` function, which is described in the previous section.

GetTrackID

The `GetTrackID` function allows your application to determine a track's unique track ID value. This ID value remains unique throughout the life of the movie.

```
pascal long GetTrackID (Track theTrack);
```

`theTrack` Specifies the track for this operation. Your application obtains this track identifier from such Movie Toolbox functions as `NewMovieTrack` and `GetMovieTrack` (described on page 2-151 and page 2-204, respectively).

DESCRIPTION

The `GetTrackID` function returns the track's ID value. If the function could not determine the ID value, it sets this returned value to 0.

ERROR CODES

`invalidTrack`	–2009	This track is corrupted or invalid

GetTrackMovie

The `GetTrackMovie` function allows you to determine the movie that contains a specified track.

```
pascal Movie GetTrackMovie (Track theTrack);
```

`theTrack` Specifies the track for this operation. Your application obtains this track identifier from such Movie Toolbox functions as `NewMovieTrack` and `GetMovieTrack` (described on page 2-151 and page 2-204, respectively).

DESCRIPTION

The `GetTrackMovie` function returns the movie identifier that corresponds to the movie that contains the track. If the function could not locate the movie, it sets this returned value to `nil`.

ERROR CODES

`invalidTrack`	–2009	This track is corrupted or invalid

GetTrackMedia

The `GetTrackMedia` function allows you to determine the media that contains a track's sample data.

```
pascal Media GetTrackMedia (Track theTrack);
```

`theTrack` Specifies the track for this operation. Your application obtains this track identifier from such Movie Toolbox functions as `NewMovieTrack` and `GetMovieTrack` (described on page 2-151 and page 2-204, respectively).

DESCRIPTION

The `GetTrackMedia` function returns the media identifier that corresponds to the media that specifies the track's sample data. If the function could not locate the media, it sets this returned value to `nil`.

ERROR CODES

`invalidTrack`	–2009	This track is corrupted or invalid

GetMediaTrack

The `GetMediaTrack` function allows you to determine the track that uses a specified media.

```
pascal Track GetMediaTrack (Media theMedia);
```

`theMedia` Specifies the media for this operation. Your application obtains this media identifier from such Movie Toolbox functions as `NewTrackMedia` and `GetTrackMedia` (described on page 2-153 and page 2-206, respectively).

DESCRIPTION

The `GetMediaTrack` function returns the track identifier of the track that uses the media. If the function cannot determine the track that uses the media, it sets this value to `nil`.

ERROR CODES

`invalidMedia`	–2008	This media is corrupted or invalid

Working With Alternate Tracks

The Movie Toolbox allows you to define alternate tracks in a movie. You can use alternate tracks to support multiple languages or to present different levels of visual quality in the movie. You collect alternate tracks into groups. Alternate track groups are collections of tracks that conceptually represent some data but are appropriate for use in different play environments. For example, you might have some 4-bit data in one track and some 8-bit data in another. Working with alternate tracks allows you to set up alternatives from which the Movie Toolbox can choose.

The Movie Toolbox selects one track from each alternate group when it plays the movie. For example, you could create a movie that has three separate audio tracks: one in English, one in French, and one in Spanish. You would collect these audio tracks into an alternate group. When the user plays the movie, the Movie Toolbox selects the track from this group that corresponds to the current language setting for the movie.

Similarly, you can use alternate tracks to store data of different quality. When the user plays the movie, the Movie Toolbox selects the track that best suits the capabilities of the Macintosh computer on which the movie is being played. In this manner, you can create a single movie that can accommodate the playback characteristics of a number of different computer configurations.

The Movie Toolbox allows you to store quality information for media structures that are assigned to either sound or video tracks. For all tracks, the Movie Toolbox uses bits 6 and 7 of the quality setting. These bits encode a relative quality value. These values range from 0 to 3. You can use higher quality values to indicate larger sample sizes. For example, consider a movie that has two sound tracks that are alternates for each other—one contains 8-bit sound while the other contains 16-bit sound. You could assign a quality value of `mediaQualityNormal` to the 8-bit media and a value of `mediaQualityBetter` to the 16-bit media. The Movie Toolbox would only play the 16-bit media if the Macintosh configuration could handle 16-bit sound. Otherwise, the Movie Toolbox would use the 8-bit media. The sound media handler determines the sample size for each sound media for the Movie Toolbox by examining the media's sound description structure.

In addition, the Movie Toolbox also uses bits 0 through 5 (the low-order bits) of the quality setting. You use these bits to indicate the pixel depths at which the media should be played. Each bit corresponds to a single depth value, ranging from 1-bit pixels to 32-bit pixels. You may use these bits to control the playback of both video and sound tracks.

As an example, consider a movie that contains three video tracks with the following characteristics:

Track A	1-bit video data, no compression
Track B	Compressed using the Apple Video Compressor
Track C	Compressed using the Joint Photographic Experts Group (JPEG) compressor

You could assign the following quality values to these track's media structures:

Track A `mediaQualityDraft` + 1-bit depth + 2-bit depth (quality value is 0x0003: 0x0000 + 0x0003)

Track B `mediaQualityNormal` + 4-bit depth + 8-bit depth + 16-bit depth + 32-bit depth (quality value is 0x007C: 0x0040 + 0x003C)

Track C `mediaQualityBetter` + 4-bit depth + 8-bit depth + 16-bit depth + 32-bit depth (quality value is 0x00BC: 0x0080 + 0x003C)

The Movie Toolbox would always use Track A when playing the movie on 1-bit and 2-bit displays. At the other pixel depths, the video media handler determines which track to use by examining the availability and performance of the specified decompressors. If the JPEG decompressor can play back at full frame rate, the Movie Toolbox would use Track C. Otherwise, the Toolbox uses Track B. The video media handler determines the compressor that is appropriate for each media by examining the media's image description structure.

You set a movie's language by calling the `SetMovieLanguage` function.

To establish alternate groups of tracks, you can use the `SetTrackAlternate` and `GetTrackAlternate` functions.

You can work with the language and quality characteristics of media by calling the `GetMediaLanguage`, `SetMediaLanguage`, `GetMediaQuality`, and `SetMediaQuality` functions.

By default, the Movie Toolbox automatically selects the appropriate tracks to play according to a movie's quality and language settings, as well as the capabilities of the Macintosh computer. Whenever your application calls the `SetMovieGWorld`, `SetMovieBox`, `UpdateMovie`, or `SetMovieMatrix` function (described on page 2-159, page 2-161, page 2-126, and page 2-170, respectively), the Movie Toolbox checks each alternate group for an appropriate track. However, you can control this selection process. Use the `SetAutoTrackAlternatesEnabled` function to enable or disable automatic track selection. The `SelectMovieAlternates` function instructs the Movie Toolbox to select appropriate tracks immediately. If no tracks in an alternate track group are enabled, then the Movie Toolbox does not activate any track from that group during automatic track selection.

SetMovieLanguage

The `SetMovieLanguage` function allows your application to specify a movie's language. You specify the language by supplying the appropriate language or region code (see *Inside Macintosh: Text* for more information on language and region codes).

```
pascal void SetMovieLanguage (Movie theMovie, long language);
```

`theMovie` Specifies the movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

`language` Specifies the movie's language or region code.

DESCRIPTION

The Movie Toolbox examines the movie's alternate groups and selects and enables appropriate tracks. If the Movie Toolbox cannot find an appropriate track, it does not change the movie's language.

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid

SelectMovieAlternates

The `SelectMovieAlternates` function allows your application to instruct the Movie Toolbox to select appropriate tracks immediately.

```
pascal void SelectMovieAlternates (Movie theMovie);
```

`theMovie` Specifies the movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

DESCRIPTION

You can call the `SelectMovieAlternates` function even if you have disabled automatic track selection with the `SetAutoTrackAlternatesEnabled` function (which is described in the next section) or by setting the `newMovieDontAutoAlternate` flag when you created the movie (see page 2-91 for details on this flag).

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid

SetAutoTrackAlternatesEnabled

The SetAutoTrackAlternatesEnabled function allows your application to enable and disable automatic track selection by the Movie Toolbox.

```
pascal void SetAutoTrackAlternatesEnabled (Movie theMovie,
                                           Boolean enable);
```

theMovie Specifies the movie for this operation. Your application obtains this movie identifier from such functions as NewMovie, NewMovieFromFile, and NewMovieFromHandle (described on page 2-92, page 2-88, and page 2-90, respectively).

enable Controls automatic track selection. Set this parameter to true to enable automatic track selection. Set this parameter to false to disable automatic track selection.

DESCRIPTION

If automatic track selection is enabled, the Movie Toolbox selects appropriate tracks whenever your application calls the SetMovieGWorld, SetMovieBox, UpdateMovie, or SetMovieMatrix functions (described on page 2-159, page 2-161, page 2-126, and page 2-170, respectively). When you enable automatic track selection, the Movie Toolbox immediately selects enabled tracks for the movie. This overrides the setting of the newMovieDontAutoAlternate flag (see page 2-91 for details on this flag).

ERROR CODES

invalidMovie	–2010	This movie is corrupted or invalid

SEE ALSO

You can instruct the Movie Toolbox to select appropriate tracks immediately by calling the SelectMovieAlternates function, which is described in the previous section.

SetTrackAlternate

The SetTrackAlternate function allows your application to add tracks to or remove tracks from alternate groups.

```
pascal void SetTrackAlternate (Track theTrack, Track alternateT);
```

theTrack Specifies the track and group for this operation. Your application obtains this track identifier from such Movie Toolbox functions as NewMovieTrack and GetMovieTrack (described on page 2-151 and

page 2-204, respectively). The SetTrackAlternate function changes this track's group affiliation based on the value of the alternateT parameter.

alternateT

Controls whether the function adds the track to a group or removes it from a group. If the alternateT parameter contains a valid track identifier, the Movie Toolbox adds this track to the group that contains the track specified by the parameter theTrack. Note that if the track identified by the parameter alternateTrack already belongs to a group, the Movie Toolbox combines the two groups into a single group.

Set this parameter to nil to remove the track specified by the theTrack parameter from its group.

ERROR CODES

invalidTrack	–2009	This track is corrupted or invalid

SEE ALSO

You can determine all the tracks in a group by calling the GetTrackAlternate function, which is described in the next section.

GetTrackAlternate

The GetTrackAlternate function allows your application to determine all the tracks in an alternate group. You specify the group by identifying a track in the group. The group list is circular, so you must specify a different track in the group each time you call this function.

```
pascal Track GetTrackAlternate (Track theTrack);
```

theTrack Specifies the track for this operation. Your application obtains this track identifier from such Movie Toolbox functions as NewMovieTrack and GetMovieTrack (described on page 2-151 and page 2-204, respectively).

DESCRIPTION

The GetTrackAlternate function returns the track identifier of the next track in the group. If the track you specify does not belong to a group, the function returns the same identifier you supply. Because the alternate group list is circular, you have retrieved all the tracks in the group when the function returns the track identifier that you supplied the first time you called the GetTrackAlternate function. If there is only one track in an alternate group, this function returns the track identifier you supply.

ERROR CODES

`invalidTrack`	–2009	This track is corrupted or invalid

SEE ALSO

You can add a track to a group by calling the `SetTrackAlternate` function, which is described in the previous section.

SetMediaLanguage

The `SetMediaLanguage` function sets a media's language or region code. You should call this function only when you are creating a new media. See *Inside Macintosh: Text* for more information on language and region codes.

```
pascal void SetMediaLanguage (Media theMedia, short language);
```

`theMedia` Specifies the media for this operation. Your application obtains this media identifier from such Movie Toolbox functions as `NewTrackMedia` and `GetTrackMedia` (described on page 2-151 and page 2-204, respectively).

`language` Specifies the media's language or region code.

ERROR CODES

`invalidMedia`	–2008	This media is corrupted or invalid

SEE ALSO

You can retrieve a media's language or region code by calling the `GetMediaLanguage` function, which is described in the next section.

GetMediaLanguage

The `GetMediaLanguage` function returns a media's language or region code. See *Inside Macintosh: Text* for more information on language and region codes.

```
pascal short GetMediaLanguage (Media theMedia);
```

`theMedia` Specifies the media for this operation. Your application obtains this media identifier from such Movie Toolbox functions as `NewTrackMedia` and `GetTrackMedia` (described on page 2-153 and page 2-206, respectively).

ERROR CODES

`invalidMedia`	–2008	This media is corrupted or invalid

SEE ALSO

You can set a media's language or region code by calling the `SetMediaLanguage` function, which is described in the previous section.

SetMediaQuality

The `SetMediaQuality` function sets a media's quality level value. The Movie Toolbox uses this quality value to determine which track it selects to play on a given Macintosh computer. You should set this value only when you are creating a new media.

```
pascal void SetMediaQuality (Media theMedia, short quality);
```

`theMedia` Specifies the media for this operation. Your application obtains this media identifier from such Movie Toolbox functions as `NewTrackMedia` and `GetTrackMedia` (described on page 2-153 and page 2-206, respectively).

`quality` Specifies the media's quality value. The quality value indicates the pixel depths at which the media can be played. This even applies to sound media. The low-order 6 bits of the quality value correspond to specific pixel depths. If a bit is set to 1, the media can be played at the corresponding depth. More than one of these bits may be set to 1. The following bits are defined:

Bit 0 1 bit per pixel

Bit 1 2 bits per pixel

Bit 2 4 bits per pixel

Bit 3 8 bits per pixel

Bit 4 16 bits per pixel

Bit 5 32 bits per pixel

In addition, bits 6 and 7 define the media's quality level. A value of 0 corresponds to the lowest quality level; a value of 3 corresponds to the highest quality level. The following constants define these values:

`mediaQualityDraft`
Specifies the lowest quality level. This constant sets bits 6 and 7 to a value of 0.

`mediaQualityNormal`
Specifies an acceptable quality level. This constant sets bits 6 and 7 to a value of 1.

`mediaQualityBetter`
Specifies a higher quality level. This constant sets bits 6 and 7 to a value of 2.

mediaQualityBest
Specifies the highest quality level. This constant sets bits 6 and 7 to a value of 3.

ERROR CODES

invalidMedia	–2008	This media is corrupted or invalid

SEE ALSO

You can retrieve the quality value of a media by calling the GetMediaQuality function, which is described in the next section.

GetMediaQuality

The GetMediaQuality function returns a media's quality level value. The Movie Toolbox uses this quality value to influence which track it selects to play on a given Macintosh computer.

```
pascal short GetMediaQuality (Media theMedia);
```

theMedia Specifies the media for this operation. Your application obtains this media identifier from such Movie Toolbox functions as NewTrackMedia and GetTrackMedia (described on page 2-153 and page 2-206, respectively).

DESCRIPTION

The GetMediaQuality function returns the media's quality value. The quality value indicates the pixel depths at which the media can be played. This even applies to sound media. The low-order 6 bits of the quality value correspond to specific pixel depths. If a bit is set to 1, the media can be played at the corresponding depth. More than one of these bits may be set to 1. The following bits are defined:

Bit 0 1 bit per pixel

Bit 1 2 bits per pixel

Bit 2 4 bits per pixel

Bit 3 8 bits per pixel

Bit 4 16 bits per pixel

Bit 5 32 bits per pixel

In addition, bits 6 and 7 define the media's quality level. A value of 0 corresponds to the lowest quality level; a value of 3 corresponds to the highest quality level.

mediaQualityDraft
Specifies the lowest quality level. This constant sets bits 6 and 7 to a value of 0.

`mediaQualityNormal`
Specifies an acceptable quality level. This constant sets bits 6 and 7 to a value of 1.

`mediaQualityBetter`
Specifies a higher quality level. This constant sets bits 6 and 7 to a value of 2.

`mediaQualityBest`
Specifies the highest quality level. This constant sets bits 6 and 7 to a value of 3.

ERROR CODES

`invalidMedia`	–2008	This media is corrupted or invalid

SEE ALSO

You can set the quality value of a media by calling the `SetMediaQuality` function, which is described in the previous section.

Working With Data References

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 (see *Inside Macintosh: Files* for more information about aliases and the Alias Manager). The type value for data references that use aliases is `'alis'`. Note that the Movie Toolbox uses aliases even on Macintosh computers that do not have System 7 installed—your application can use Alias Manager routines if the Movie Toolbox is installed. See “The Movie Toolbox and System 6” on page 2-63 for more information.

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.

AddMediaDataRef

The AddMediaDataRef function adds a data reference to a media.

```
pascal OSErr AddMediaDataRef (Media theMedia, short *index,
                              Handle dataRef,
                              OSType dataRefType);
```

theMedia Specifies the media for this operation. Your application obtains this media identifier from such Movie Toolbox functions as NewTrackMedia and GetTrackMedia (described on page 2-153 and page 2-206, respectively).

index Contains a pointer to a short integer. The Movie Toolbox returns the index value that is assigned to the new data reference. Your application can use this index to identify the reference to other Movie Toolbox functions, such as GetMediaDataRef (described on page 2-217). If the Movie Toolbox cannot add the data reference to the media, it sets the returned index value to 0.

dataRef Specifies the data reference. This parameter contains a handle to the information that identifies the file that contains this media's data. The type of information stored in that handle depends upon the value of the dataRefType parameter.

dataRefType
Specifies the type of data reference. If the data reference is an alias, you must set this parameter to rAliasType ('alis'), indicating that the reference is an alias. See *Inside Macintosh: Files* for more information about aliases and the Alias Manager.

ERROR CODES

invalidMedia	–2008	This media is corrupted or invalid

SetMediaDataRef

The SetMediaDataRef function changes the file that the specified media identifies as the location for its data storage.

```
pascal OSErr SetMediaDataRef (Media themedia, short index,
                              Handle dataRef, OSType dataRefType);
```

themedia Specifies the media for this operation. Your application obtains this media identifier from such Movie Toolbox functions as NewTrackMedia and GetTrackMedia (described on page 2-153 and page 2-206, respectively).

index Contains a pointer to a short integer. The Movie Toolbox returns the index value that is assigned to the new data reference. Your application can use this index to identify the reference to other Movie Toolbox functions, such

as GetMediaDataRef (described on page 2-217). As with all data reference functions, the index starts with 1. If the Movie Toolbox cannot add the data reference to the media, it sets the returned index value to 0.

dataRef Specifies the data reference. This parameter contains a handle to the information that identifies the file that contains this media's data. The type of information stored in that handle depends upon the value of the dataRefType parameter.

dataRefType
Specifies the type of data reference. If the data reference is an alias, you must set this parameter to rAliasType ('alis'), indicating that the reference is an alias. See *Inside Macintosh: Files* for more information about aliases and the Alias Manager.

SPECIAL CONSIDERATIONS

Don't call this function unless you have a really good reason. However, if you want to resolve your own missing data references, or you are developing a special-purpose kind of application, SetMediaDataRef may be quite useful.

GetMediaDataRef

The GetMediaDataRef function returns a copy of a specified data reference. Your application identifies the data reference with the appropriate data reference index.

```
pascal OSErr GetMediaDataRef (Media theMedia, short index,
                              Handle *dataRef,
                              OSType *dataRefType,
                              long *dataRefattributes);
```

theMedia Specifies the media for this operation. Your application obtains this media identifier from such Movie Toolbox functions as NewTrackMedia and GetTrackMedia (described on page 2-153 and page 2-206, respectively).

index Identifies the data reference. You provide the index value that corresponds to the data reference. It must be less than or equal to the value that is returned by the GetMediaDataRefCount function, described in the previous section.

dataRef Contains a pointer to a field that is to receive a handle to the data reference. The media handler returns a handle to information that identifies the file that contains this media's data. The type of information stored in that handle depends upon the value of the dataRefType parameter. If the function cannot locate the specified data reference, the handler sets this returned value to nil. Set the dataRef parameter to nil if you are not interested in this information.

dataRefType
: Contains a pointer to a field that is to receive the type of data reference. If the data reference is an alias, the media handler sets this value to 'alis', indicating that the reference is an alias. Set the dataRefType parameter to nil if you are not interested in this information.

dataRefattributes
: Contains a pointer to a field that is to receive the reference's attribute flags. The following flags are available (unused flags are set to 0):

  dataRefSelfReference
  : Indicates whether the data reference refers to the movie resource's data file. If this flag is set to 1, the data reference identifies media data that is stored in the same file as the movie resource.

  dataRefWasNotResolved
  : Indicates whether the Movie Toolbox resolved the data reference. If this flag is set to 1, the Movie Toolbox could not resolve the data reference. For example, the toolbox may be unable to resolve data references because the required storage device is unavailable at the time a movie is loaded. If the data reference is unresolved, the Movie Toolbox disables the corresponding track.

  Set the dataRefAttributes parameter to nil if you are not interested in this information.

DESCRIPTION

You can use GetMediaDataRef function to retrieve information about a data reference. For example, you might want to verify the condition of a movie's data references after loading the movie from its movie file. You could use this function to check each data reference.

ERROR CODES

invalidMedia	–2008	This media is corrupted or invalid

SEE ALSO

You can add a data reference to a media by calling the AddMediaDataRef function, which is described on page 2-216. You must dispose of a media's data references yourself by disposing of its handle. You can determine the number of data references in a media by calling the GetMediaDataRefCount function, which is described in the previous section.

GetMediaDataRefCount

The `GetMediaDataRefCount` function allows your application to determine the number of data references in a media.

```
pascal OSErr GetMediaDataRefCount (Media theMedia, short *count);
```

`theMedia` Specifies the media for this operation. Your application obtains this media identifier from such Movie Toolbox functions as `NewTrackMedia` and `GetTrackMedia` (described on page 2-153 and page 2-206, respectively).

`count` Contains a pointer to a field that is to receive the number of data references in the media.

DESCRIPTION

The count of references in a media corresponds to the maximum index value of any reference in the media. You can use this value to control a loop in which you retrieve all of a media's data references, using the `GetMediaDataRef` function, which is described in the next section.

ERROR CODES

`invalidMedia`	–2008	This media is corrupted or invalid

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. This section describes those functions.

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.

GetMovieCreationTime

The `GetMovieCreationTime` function returns a long integer that contains the movie's creation date and time information.

```
pascal unsigned long GetMovieCreationTime (Movie theMovie);
```

`theMovie` Specifies the movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid

GetMovieModificationTime

The `GetMovieModificationTime` function returns a movie's modification date.

```
pascal unsigned long GetMovieModificationTime (Movie theMovie);
```

`theMovie` Specifies the movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

DESCRIPTION

The `GetMovieModificationTime` function returns a long integer that contains the movie's modification date and time information.

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid

GetTrackCreationTime

The `GetTrackCreationTime` function returns a track's creation date.

```
pascal unsigned long GetTrackCreationTime (Track theTrack);
```

theTrack Specifies the track for this operation. Your application obtains this track identifier from such Movie Toolbox functions as NewMovieTrack and GetMovieTrack (described on page 2-151 and page 2-204, respectively).

DESCRIPTION

The GetTrackCreationTime function returns a long integer that contains the track's creation date and time information.

ERROR CODES

invalidTrack	–2009	This track is corrupted or invalid

GetTrackModificationTime

The GetTrackModificationTime function returns a track's modification date.

```
pascal unsigned long GetTrackModificationTime (Track theTrack);
```

theTrack Specifies the track for this operation. Your application obtains this track identifier from such Movie Toolbox functions as NewMovieTrack and GetMovieTrack (described on page 2-151 and page 2-204, respectively).

DESCRIPTION

The GetTrackModificationTime function returns a long integer that contains the track's modification date and time information.

ERROR CODES

invalidTrack	–2009	This track is corrupted or invalid

GetMediaCreationTime

The GetMediaCreationTime function returns the creation date stored in the media.

```
pascal unsigned long GetMediaCreationTime (Media theMedia);
```

theMedia Specifies the media for this operation. Your application obtains this media identifier from such Movie Toolbox functions as NewTrackMedia and GetTrackMedia (described on page 2-153 and page 2-206, respectively).

DESCRIPTION

The `GetMediaCreationTime` function returns a long integer that contains the media's creation date and time information.

ERROR CODES

`invalidMedia`	–2008	This media is corrupted or invalid

GetMediaModificationTime

The `GetMediaModificationTime` function returns a media's modification date.

```
pascal unsigned long GetMediaModificationTime (Media theMedia);
```

`theMedia` Specifies the media for this operation. Your application obtains this media identifier from such Movie Toolbox functions as `NewTrackMedia` and `GetTrackMedia` (described on page 2-153 and page 2-206, respectively).

DESCRIPTION

The `GetMediaModificationTime` function returns a long integer that contains the media's modification date and time information.

ERROR CODES

`invalidMedia`	–2008	This media is corrupted or invalid

Working With Media Samples

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. Refer to "Adding Samples to Media Structures" beginning on page 2-271 for information about functions that allow you to retrieve sample data from a media.

Your application can use the `GetMovieDataSize`, `GetTrackDataSize`, 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" beginning on page 2-196 to locate specific samples in a media.

GetMovieDataSize

The `GetMovieDataSize` function allows your application to determine the size, in bytes, of the sample data in a segment of a movie.

```
pascal long GetMovieDataSize (Movie theMovie, TimeValue startTime,
                              TimeValue duration);
```

`theMovie` Specifies the movie for this operation. You obtain this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

`startTime` Contains a time value specifying the starting point of the segment.

`duration` Contains a time value that specifies the duration of the segment.

DESCRIPTION

The `GetMovieDataSize` function returns a long integer that contains the size, in bytes, of the movie's sample data that lies in the specified segment. `GetMovieDataSize` counts each use of a sample. That is, if a movie uses a given sample more than once, the size of that sample is included in the returned size value one time for each use. Consequently, the returned size is greater than or equal to the actual size of the movie's sample data, and corresponds to the amount of movie data that will be retrieved when you call the `FlattenMovie` function or `FlattenMovieData` function (described on page 2-105 and page 2-107, respectively).

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid
`invalidDuration`	–2014	This duration value is invalid
`invalidTime`	–2015	This time value is invalid

GetTrackDataSize

The `GetTrackDataSize` function allows your application to determine the size, in bytes, of the sample data in a segment of a track.

```
pascal long GetTrackDataSize (Track theTrack, TimeValue startTime,

                              TimeValue duration);
```

`theTrack` Specifies the track for this operation. You obtain this track identifier from such Movie Toolbox functions as `NewMovieTrack` and `GetMovieTrack` (described on page 2-151 and page 2-204, respectively).

`startTime` Contains a time value specifying the starting point of the segment.

`duration` Contains a time value that specifies the duration of the segment.

DESCRIPTION

The `GetTrackDataSize` function returns a long integer that contains the size, in bytes, of the track's sample data that lies in the specified segment.

This function counts each use of a sample. That is, if a track uses a given sample more than once, the size of that sample is included in the returned size value one time for each use. Consequently, the returned size is greater than or equal to the actual size of the track's sample data.

ERROR CODES

`invalidTrack`	–2009	This track is corrupted or invalid
`invalidDuration`	–2014	This duration value is invalid
`invalidTime`	–2015	This time value is invalid

GetMediaDataSize

The `GetMediaDataSize` function allows your application to determine the size, in bytes, of the sample data in a media segment.

```
pascal long GetMediaDataSize (Media theMedia, TimeValue startTime,
                              TimeValue duration);
```

`theMedia` Specifies the media for this operation. Your application obtains this media identifier from such Movie Toolbox functions as `NewTrackMedia` and `GetTrackMedia` (described on page 2-153 and page 2-206, respectively).

`startTime` Contains a time value specifying the starting point of the segment.

`duration` Contains a time value that specifies the duration of the segment.

DESCRIPTION

The `GetMediaDataSize` function returns a long integer that contains the size, in bytes, of the media's sample data that lies in the specified segment. Note that this number does not necessarily correspond to the amount of sample data used in the track that contains the media. Some samples in the media may not be used in the track, and others may be used more than once.

ERROR CODES

`invalidMedia`	–2008	This media is corrupted or invalid
`invalidDuration`	–2014	This duration value is invalid
`invalidTime`	–2015	This time value is invalid

GetMediaSampleCount

The `GetMediaSampleCount` function allows you to determine the number of samples in a media.

```
pascal long GetMediaSampleCount (Media theMedia);
```

`theMedia` Specifies the media for this operation. You obtain this media identifier from such Movie Toolbox functions as `NewTrackMedia` and `GetTrackMedia` (described on page 2-153 and page 2-206, respectively).

DESCRIPTION

The `GetMediaSampleCount` function returns a long integer that contains the number of samples in the specified media. Note that this number does not necessarily correspond to the number of samples used in the track that contains the media. Some samples in the media may not be used in the track, and others may be used more than once.

ERROR CODES

`invalidMedia`	–2008	This media is corrupted or invalid

GetMediaSampleDescriptionCount

The `GetMediaSampleDescriptionCount` function returns the number of sample descriptions in a media.

```
pascal long GetMediaSampleDescriptionCount (Media theMedia);
```

`theMedia` Specifies the media for this operation. You obtain this media identifier from such Movie Toolbox functions as `NewTrackMedia` and `GetTrackMedia` (described on page 2-153 and page 2-206, respectively).

DESCRIPTION

The Movie Toolbox identifies a media's sample descriptions with an index value. Index values always range from 1 to the number of sample descriptions in the media. Sample description indexes provide a convenient way to access each sample description in a media.

The format of sample descriptions differs by media type. Sample descriptions for image data are defined by image description structures, which are discussed in the chapter "Image Compression Manager" in this book. Sample descriptions for sound are defined by sound description structures, which are discussed in "The Sound Description Structure" beginning on page 2-79. Sample descriptions for text are defined by text

description structures, which are described in “Text Media Handler Functions” beginning on page 2-290.

ERROR CODES

`invalidMedia`	–2008	This media is corrupted or invalid

SEE ALSO

You can use the value returned by this function to control a loop in which you retrieve each sample description in a media by calling the `GetMediaSampleDescription` function, which is described in the next section.

GetMediaSampleDescription

The `GetMediaSampleDescription` function allows you to retrieve a sample description from a media.

```
pascal void GetMediaSampleDescription (Media theMedia, long index,
                                       SampleDescriptionHandle descH);
```

`theMedia`	Specifies the media for this operation. You obtain this media identifier from such Movie Toolbox functions as `NewTrackMedia` and `GetTrackMedia` (described on page 2-153 and page 2-206, respectively).
`index`	Specifies the index of the sample description to retrieve. This index corresponds to the sample description itself, not the samples in the media.
`descH`	Specifies a handle that is to receive the sample description. The Movie Toolbox correctly resizes this handle for the returned sample description. If there is no description for the specified index, the function returns this handle unchanged. Your application must allocate and dispose of this handle.

DESCRIPTION

This function provides a convenient way to retrieve information that describes a sample. For example, you can use this function to retrieve an image media’s color lookup table.

The format of sample descriptions differs by media type. Sample descriptions for image data are defined by image description structures, which are discussed in the chapter “Image Compression Manager” in this book. Sample descriptions for sound are defined by sound description structures, which are discussed earlier in this chapter. Sample descriptions for text are defined by text description data structures, which are described in “Text Media Handler Functions” beginning on page 2-290.

The Movie Toolbox identifies a media's sample descriptions with an index value. Index values always range from 1 to the number of sample descriptions in the media. Sample description indexes provide a convenient way to access each sample description in a media.

ERROR CODES

`invalidMedia`	–2008	This media is corrupted or invalid
`badDataRefIndex`	–2050	Data reference index value is invalid

Memory Manager errors

SEE ALSO

You can determine the number of sample descriptions in a media by calling the `GetMediaSampleDescriptionCount` function, which is described in the previous section.

SetMediaSampleDescription

The `SetMediaSampleDescription` function lets you change the contents of a particular sample description of a specified media.

```
pascal OSErr SetMediaSampleDescription (Media theMedia,
                                        long index,
                                    SampleDescriptionHandle descH);
```

`theMedia`	Specifies the media for this operation. You obtain this media identifier from such Movie Toolbox functions as `NewTrackMedia` and `GetTrackMedia` (described on page 2-153 and page 2-206, respectively).
`index`	Specifies the index of the sample description to be changed. This index corresponds to the sample description itself, not the samples in the media. This long integer must be between 1 and the largest sample description index.
`descH`	Specifies the handle to the sample description. If there is no description for the specified index, the function returns this handle unchanged.

DESCRIPTION

The `SetMediaSampleDescription` function can be useful in the case of a media handler, such as a text media handler, that stores playback information in its sample description, as opposed to just data format information (as in the case of the video media handler). For more on media handlers, see *Inside Macintosh: QuickTime Components*.

SPECIAL CONSIDERATIONS

Because a sample description structure may define the format of the data, you should not assume the description describes the data. You should use this function only on an inactive track.

MediaTimeToSampleNum

The `MediaTimeToSampleNum` function allows you to find the sample that contains the data for a specified time. You indicate the time in the media's time scale.

```
pascal void MediaTimeToSampleNum (Media theMedia, TimeValue time,
                                  long *sampleNum,
                                  TimeValue *sampleTime,
                                  TimeValue *sampleDuration);
```

`theMedia` Specifies the media for this operation. You obtain this media identifier from such Movie Toolbox functions as `NewTrackMedia` and `GetTrackMedia` (described on page 2-153 and page 2-206, respectively).

`time` Specifies the time for which you are retrieving sample information. You must specify this value in the media's time scale.

`sampleNum` Contains a pointer to a long integer that is to receive the sample number. The Movie Toolbox returns the sample number that identifies the sample that contains data for the time specified by the `time` parameter.

`sampleTime`
Contains a pointer to a time value. The `MediaTimeToSampleNum` function updates this time value to indicate the starting time of the sample that contains data for the time specified by the `time` parameter. This time value is expressed in the media's time scale. Set this parameter to `nil` if you do not want this information.

`sampleDuration`
Contains a pointer to a time value. The Movie Toolbox returns the duration of the sample that contains data for the time specified by the `time` parameter. This time value is expressed in the media's time scale. Set this parameter to `nil` if you do not want this information.

DESCRIPTION

The Movie Toolbox returns information about the sample that contains data for that time, including its starting time, duration, and sample number.

The `MediaTimeToSampleNum` function does not account for edits applied to the media by a movie's tracks. If you want to work with edits, use the functions that allow you to look for interesting times. These functions are described in "Finding Interesting Times," beginning on page 2-196.

ERROR CODES

`invalidMedia`	–2008	This media is corrupted or invalid

SEE ALSO

You can convert a sample number into a time in a media's time scale by calling the `SampleNumToMediaTime` function, which is described in the next section.

SampleNumToMediaTime

The `SampleNumToMediaTime` function allows you to find the time at which a specified sample plays. This time is expressed in the media's time scale.

```
pascal void SampleNumToMediaTime (Media theMedia,
                                  long logicalSampleNum,
                                  TimeValue *sampleTime,
                                  TimeValue *sampleDuration);
```

`theMedia` Specifies the media for this operation. You obtain this media identifier from such Movie Toolbox functions as `NewTrackMedia` and `GetTrackMedia` (described on page 2-153 and page 2-206, respectively).

`logicalSampleNum`
Specifies the sample number.

`sampleTime`
Contains a pointer to a time value. The `MediaTimeToSampleNum` function updates this time value to indicate the starting time of the sample specified by the `logicalSampleNum` parameter. This time value is expressed in the media's time scale. Set this parameter to `nil` if you do not want this information.

`sampleDuration`
Contains a pointer to a time value. The Movie Toolbox returns the duration of the sample specified by the `logicalSampleNum` parameter. This time value is expressed in the media's time scale. Set this parameter to `nil` if you do not want this information.

ERROR CODES

`invalidMedia`	–2008	This media is corrupted or invalid

SEE ALSO

You can find the sample for a specified time by calling the `MediaTimeToSampleNum` function, which is described in the previous section.

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 © character (Option-G) to specify user data items that store text data.

The following user data types are currently defined:

`'©nam'`	Movie's name
`'©cpy'`	Copyright statement
`'©day'`	Date the movie content was created
`'©dir'`	Name of movie's director
`'©ed1' to '©ed9'`	Edit dates and descriptions
`'©fmt'`	Indication of movie format (computer-generated, digitized, and so on)
`'©inf'`	Information about the movie
`'©prd'`	Name of movie's producer
`'©prf'`	Names of performers
`'©req'`	Special hardware and software requirements
`'©src'`	Credits for those who provided movie source content
`'©wrt'`	Name of movie's writer

User data items of these types must contain text data only.

Second, the Movie Toolbox allows you to create more than one user data item in a user data list. Therefore, each user data item is identified by a unique index. Index values are assigned sequentially within a user data type and start at 1.

Finally, you may create alternate text for a given user data text item. For example, you may want to support multiple languages and may therefore want to create different text for each language. The Movie Toolbox allows you to specify different versions of the text of a single user data item. These versions are distinguished by their region code values.

The Movie Toolbox provides a number of functions that allow you to work with user data. Before you can work with the contents of a user data list, you must obtain a reference to the list. The `GetMovieUserData`, `GetTrackUserData`, or `GetMediaUserData` functions allow you to get a reference to a user data list. You can then use the `GetUserData`, `AddUserData`, and `RemoveUserData` functions to work with the items contained in the user data list. If your user data items contain text data, you can use the `AddUserDataText`, `GetUserDataText`, and `RemoveUserDataText` functions to work with the text of a user data item. Note that a single user data item can store either text or other data, but not both.

You can count the number of user data items of a specified type in a movie, track, or media by calling the `CountUserDataType` function. You can use the `GetNextUserDataType` function to scan all the types of user data in a specified user data list.

The Movie Toolbox also supplies a number of functions for the manipulation of user data. The `SetUserDataItem` and `GetUserDataItem` functions allow easy access to data stored in user data items. The `NewUserData` and `DisposeUserData` functions provide for the use of user data outside of the immediate context of QuickTime movies. Your applications and components can also create user data structures. The `PutUserDataIntoHandle` and the `NewUserDataFromHandle` functions permit user data to be stored and retrieved in a manner similar to public movies (also called *atoms*). See the chapter "Movie Resource Formats" in this book for details on atoms.

GetMovieUserData

The `GetMovieUserData` function allows your application to obtain access to a movie's user data list. You can then use the `GetUserData`, `AddUserData`, and `RemoveUserData` functions (described on page 2-235, page 2-235, and page 2-236, respectively) to manipulate the contents of the user data list. If the data list contains text data, you can use the `GetUserDataText`, `AddUserDataText`, and `RemoveUserDataText` functions (described on page 2-237, page 2-236, and page 2-238, respectively) to work with its contents.

```
pascal UserData GetMovieUserData (Movie theMovie);
```

`theMovie` Specifies the movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

DESCRIPTION

The `GetMovieUserData` function returns a reference to the movie's user data list. This reference is valid until you dispose of the movie. When you save the movie, the Movie Toolbox saves the user data as well. If the function could not locate the movie's user data, it sets this returned value to `nil`.

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid

Memory Manager errors

SEE ALSO

You can use the GetMediaUserData function (described on page 2-233) to gain access to a media's user data. Similarly, you can use the GetTrackUserData function (described in the next section) to work with a track's user data.

GetTrackUserData

The GetTrackUserData function allows your application to obtain access to a track's user data list. You can then use the GetUserData, AddUserData, and RemoveUserData functions (described on page 2-235, page 2-235, and page 2-236, respectively) to manipulate the contents of the user data list. If the data list contains text data, you can use the GetUserDataText, AddUserDataText, and RemoveUserDataText functions (described on page 2-237, page 2-236, and page 2-238, respectively) to work with its contents.

```
pascal UserData GetTrackUserData (Track theTrack);
```

theTrack Specifies the track for this operation. Your application obtains this track identifier from such Movie Toolbox functions as NewMovieTrack and GetMovieTrack (described on page 2-151 and page 2-204, respectively).

DESCRIPTION

The GetTrackUserData function returns a reference to the track's user data list. This reference is valid until you dispose of the track. When you save the track, the Movie Toolbox saves the user data as well. If the function could not locate the track's user data, it sets this returned value to nil.

ERROR CODES

invalidTrack	–2009	This track is corrupted or invalid

Memory Manager errors

SEE ALSO

You can use the GetMediaUserData function to gain access to a media's user data (described on page 2-233). Similarly, you can use the GetMovieUserData function (described on page 2-231) to work with a movie's user data.

GetMediaUserData

The `GetMediaUserData` function allows your application to obtain access to a media's user data list. You can then use the `GetUserData`, `AddUserData`, and `RemoveUserData` functions (described on page 2-235, page 2-235, and page 2-236, respectively) to manipulate the contents of the user data list. If the data list contains text data, you can use the `GetUserDataText`, `AddUserDataText`, and `RemoveUserDataText` functions (described on page 2-237, page 2-236, and page 2-238, respectively) to work with its contents.

```
pascal UserData GetMediaUserData (Media theMedia);
```

`theMedia` Specifies the media for this operation. Your application obtains this media identifier from such Movie Toolbox functions as `NewTrackMedia` and `GetTrackMedia` (described on page 2-153 and page 2-206, respectively).

DESCRIPTION

The `GetMediaUserData` function returns a reference to the media's user data list. This reference is valid until you dispose of the media. When you save the media, the Movie Toolbox saves the user data as well. If the function could not locate the media's user data, it sets this returned value to `nil`.

ERROR CODES

`invalidMedia`	–2008	This media is corrupted or invalid

Memory Manager errors

SEE ALSO

You can use the `GetMovieUserData` function to gain access to a movie's user data (described on page 2-231). Similarly, you can use the `GetTrackUserData` function (described in the previous section) to work with a track's user data.

GetNextUserDataType

The `GetNextUserDataType` function allows you to retrieve the next user data type in a specified user data list. You can use this function to scan all the user data types in a user data list.

```
pascal long GetNextUserDataType (UserData theUserData,
                                 OSType udType);
```

theUserData
: Specifies the user data list for this operation. You obtain this list reference by calling the `GetMovieUserData`, `GetTrackUserData`, or `GetMediaUserData` function (described on page 2-231, page 2-232, and page 2-233, respectively).

udType
: Specifies a user data type. Set this parameter to 0 to retrieve the first user data type in the user data list. On subsequent requests, use the previous value returned by this function.

DESCRIPTION

The `GetNextUserDataType` function returns an operating-system data type containing the next user data type value in the specified user data list. When you reach the end of the user data list, this function sets the returned value to 0. You can use this value to stop your scanning loop.

ERROR CODES

None

CountUserDataType

The `CountUserDataType` function allows you to determine the number of items of a given type in a user data list.

```
pascal short CountUserDataType (UserData theUserData,
                                OSType udType);
```

theUserData
: Specifies the user data list for this operation. You obtain this list reference by calling the `GetMovieUserData`, `GetTrackUserData`, or `GetMediaUserData` function (described on page 2-231, page 2-232, and page 2-233, respectively).

udType
: Specifies the type. The Movie Toolbox determines the number of items of this type in the user data list.

DESCRIPTION

The `CountUserDataType` function returns a short integer that contains the number of items of the specified type in the user data list.

ERROR CODES

None

AddUserData

The AddUserData function allows your application to add an item to a user data list. You specify the user data list, the data to be added, and the data's type value.

```
pascal OSErr AddUserData (UserData theUserData,
                          Handle data, OSType udType);
```

theUserData
: Specifies the user data list for this operation. You obtain this item reference by calling the GetMovieUserData, GetTrackUserData, or GetMediaUserData function (described on page 2-231, page 2-232, and page 2-233, respectively).

data
: Contains a handle to the data to be added to the user data list.

udType
: Specifies the type that is to be assigned to the new item.

DESCRIPTION

The Movie Toolbox places the specified data into the user data and assigns an index value that identifies the new item.

ERROR CODES

Memory Manager errors

GetUserData

The GetUserData function returns a specified user data item.

```
pascal OSErr GetUserData (UserData theUserData, Handle data,
                          OSType udType, long index);
```

theUserData
: Specifies the user data list for this operation. You obtain this list reference by calling the GetMovieUserData, GetTrackUserData, or GetMediaUserData function (described on page 2-231, page 2-232, and page 2-233, respectively).

data
: Contains a handle that is to receive the data from the specified item. The GetUserData function resizes this handle as appropriate to accommodate the item. Your application is responsible for releasing this handle when you are done with it. Set this parameter to nil if you do not want to retrieve the user data item. This can be useful if you want to verify that a user data item exists, but you do not need to work with the item's contents.

udType
: Specifies the item's type value.

index Specifies the item's index value. This parameter must specify an item in the user data list identified by the parameter theUserData.

ERROR CODES

userDataItemNotFound	–2026	Cannot locate this user data item

Memory Manager errors

RemoveUserData

The RemoveUserData function removes an item from a user data list. After the Movie Toolbox removes the item, it renumbers the remaining items of that type so that the index values are sequential and start at 1.

```
pascal OSErr RemoveUserData (UserData theUserData, OSType udType,
                             long index);
```

theUserData
Specifies the user data list for this operation. You obtain this list reference by calling the GetMovieUserData, GetTrackUserData, or GetMediaUserData function (described on page 2-231, page 2-232, and page 2-233, respectively).

udType Specifies the item's type value.

index Specifies the item's index value. This parameter must specify an item in the user data list identified by the parameter theUserData.

ERROR CODES

userDataItemNotFound	–2026	Cannot locate this user data item

Memory Manager errors

AddUserDataText

The AddUserDataText function allows your application to place language-tagged text into an item in a user data list. You specify the user data list and item, the data to be added, the data's type value, and the language code of the data.

```
pascal OSErr AddUserDataText (UserData theUserData, Handle data,
                              OSType udType, long index,
                              short itlRegionTag);
```

theUserData
: Specifies the user data list for this operation. You obtain this list reference by calling the `GetMovieUserData`, `GetTrackUserData`, or `GetMediaUserData` function (described on page 2-231, page 2-232, and page 2-233, respectively).

data
: Contains a handle to the data to be added to the user data list.

udType
: Specifies the type that is to be assigned to the new item.

index
: Specifies the item to which the text is to be added. This parameter must specify an item in the user data list identified by the parameter `theUserData`.

itlRegionTag
: Specifies the region code of the text to be added. If there is already text with this region code in the item, the function replaces the existing text with the data specified by the `data` parameter. See *Inside Macintosh: Text* for more information about language and region codes.

DESCRIPTION

The Movie Toolbox places the specified data into the user data item. If the item does not exist when you call this function, the Movie Toolbox creates a new item for you (this is true only if the item you are adding is the first item in the list; otherwise, you must create the item yourself).

ERROR CODES

`userDataItemNotFound`	–2026	Cannot locate this user data item

Memory Manager errors

GetUserDataText

The `GetUserDataText` function allows your application to retrieve language-tagged text from an item in a user data list. You specify the user data list and item, and the item's type value and language code. The Movie Toolbox retrieves the specified text from the user data item.

```
pascal OSErr GetUserDataText (UserData theUserData, Handle data,
                              OSType udType, long index,
                              short itlRegionTag);
```

theUserData
: Specifies the user data list for this operation. You obtain this list reference by calling the `GetMovieUserData`, `GetTrackUserData`, or `GetMediaUserData` function (described on page 2-231, page 2-232, and page 2-233, respectively).

`data` Contains a handle that is to receive the data. The `GetUserDataText` function resizes this handle as appropriate. Your application must dispose of the handle when you are done with it.

`udType` Specifies the item's type value.

`index` Specifies the item's index value. This parameter must specify an item in the user data list identified by the parameter `theUserData`.

`itlRegionTag`
Specifies the language code of the text to be retrieved. See *Inside Macintosh: Text* for more information about language and region codes.

ERROR CODES

`userDataItemNotFound`	–2026	Cannot locate this user data item

Memory Manager errors

RemoveUserDataText

The `RemoveUserDataText` function allows your application to remove language-tagged text from an item in a user data list. You specify the user data list and item, and the item's type value and language code. The Movie Toolbox removes the specified text from the user data item.

```
pascal OSErr RemoveUserDataText (UserData theUserData,
                                 OSType udType, long index,
                                 short itlRegionTag);
```

`theUserData`
Specifies the user data list for this operation. You obtain this list reference by calling the `GetMovieUserData`, `GetTrackUserData`, or `GetMediaUserData` function (described on page 2-231, page 2-232, and page 2-233, respectively).

`udType` Specifies the item's type value.

`index` Specifies the item's index value. This parameter must specify an item in the user data list identified by the parameter `theUserData`.

`itlRegionTag`
Specifies the language code of the text to be removed. See *Inside Macintosh: Text* for more information about language and region codes.

ERROR CODES

`userDataItemNotFound`	–2026	Cannot locate this user data item

Memory Manager errors

SetUserDataItem

The `SetUserDataItem` allows your application to set an item in a user data list. You specify the user data list, the data to be set, the size of the data to be set, and the data's type value.

```
pascal OSErr SetUserDataItem (UserData theUserData,
                              void *data, long size, long udType,
                              long index);
```

`theUserData`
Specifies the user data list for this operation. You obtain this item reference by calling the `GetMovieUserData`, `GetTrackUserData`, or `GetMediaUserData` function (described on page 2-231, page 2-232, and page 2-233, respectively).

`data` Contains a pointer to the data item to be set in a user data list.

`size` Specifies the size of the information pointed to by the `data` parameter.

`udType` Specifies the type value assigned to the new item.

`index` Specifies the item's index value. This parameter must specify an item in the user data list identified by the parameter `theUserData`. An index value of 0 or 1 implies the first item, which is created if it doesn't already exist.

DESCRIPTION

You must provide the size of the information specified in the `data` parameter because the data may be embedded inside a larger data structure or may be on the stack.

SPECIAL CONSIDERATIONS

The data pointer must be locked, since `SetUserDataItem` may move memory.

SEE ALSO

The `SetUserDataItem` function is a pointer-based version of `AddUserData`, which is described on page 2-235.

ERROR CODES

Memory Manager errors

GetUserDataItem

The `GetUserDataItem` function returns a specified user data item. `GetUserDataItem` is a pointer-based version of the `GetUserData` function, which is described on page 2-235.

```
pascal OSErr GetUserDataItem (UserData theUserData,
                              void *data, long size,
                              OSType udType, long index);
```

`theUserData`
Specifies the user data list for this operation. You obtain this list reference by calling the `GetMovieUserData`, `GetTrackUserData`, or `GetMediaUserData` function (described on page 2-231, page 2-232, and page 2-233, respectively).

`data` Contains a pointer that is to receive the data from the specified item.

`size` Specifies the size of the item.

`udType` Specifies the item's type value.

`index` Specifies the item's index value. This parameter must specify an item in the user data list identified by the parameter `theUserData`.

DESCRIPTION

If the `size` field provided doesn't match the exact size of the actual user data item, an error is returned. In this case, you should use `GetUserData` instead. `GetUserDataItem` is useful for retrieving small, fixed-size pieces of user data without having to create a handle. You can pass 0 or 1 for the `index` parameter to indicate the first item.

ERROR CODES

`userDataItemNotFound`	–2026	Cannot locate this user data item

Memory Manager errors

NewUserData

The `NewUserData` function creates a new user data structure.

```
pascal OSErr NewUserData (UserData *theUserData);
```

`theUserData`
Contains a pointer to the user data structure.

DESCRIPTION

You can manipulate the user data structure with any of the standard user data functions described in "Working With Movie User Data" beginning on page 2-230. If the `NewUserData` function fails, the parameter `theUserData` is set to `nil`.

ERROR CODES

`memFullErr`	–108	Not enough room in heap zone

DisposeUserData

The `DisposeUserData` function disposes of a user data structure created by the `NewUserData` function.

```
pascal OSErr DisposeUserData (UserData theUserData);
```

`theUserData`
Specifies the user data structure that is to be disposed of. It is acceptable but unnecessary to pass `nil` in the parameter `theUserData`.

DESCRIPTION

You should call `DisposeUserData` only on a user data structure that you have allocated.

SPECIAL CONSIDERATIONS

Don't dispose of user data references obtained from the Movie Toolbox function `GetMovieUserData`, `GetTrackUserData`, or `GetMediaUserData` (described on page 2-231, page 2-232, and page 2-233, respectively).

PutUserDataIntoHandle

The `PutUserDataIntoHandle` function takes a specified user data structure and replaces the contents of the handle with a publicly parseable form of the user data.

```
pascal OSErr PutUserDataIntoHandle (UserData theUserData,
                                    Handle h);
```

`theUserData`
Specifies the user data structure that is to be disposed of.

`h` Contains a handler to the user data structure specified in the parameter `theUserData`.

DESCRIPTION

The contents of the h parameter are appropriate for storage as an atom, much like a public movie. See the chapter "Movie Resource Formats" in this book for details on the QuickTime atoms.

NewUserDataFromHandle

The NewUserDataFromHandle function creates a new user data structure from a handle.

```
pascal OSErr NewUserDataFromHandle (Handle h,
                                    UserData *theUserData);
```

h Contains a handle to the data structure specified in the parameter theUserData.

theUserData
Contains a pointer to a new user data structure.

DESCRIPTION

The handle specified in the h parameter must be in the standard user data storage format (that is, as an atom, just like a public movie). Usually the handle will have been created by calling PutUserDataIntoHandle (described in the previous section).

ERROR CODES

memFullErr	–108	Not enough room in heap zone

Functions for Editing Movies

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

- "Editing Movies" describes a number of functions that work with the current movie selection, supporting such user operations as cut, copy, and paste
- "Undo for Movies" discusses the functions that your application can use to support an undo capability for movie editing
- "Low-Level Movie-Editing Functions" discusses several functions that allow your application to perform detailed editing on movies
- "Editing Tracks" describes functions that your application can use to edit the contents of tracks

- "Undo for Tracks" discusses the functions that your application can use to support an undo capability for track editing
- "Adding Samples to Media Structures" describes the Movie Toolbox functions that allow you to edit media

Editing Movies

The Movie Toolbox provides a set of high-level functions that allow you to edit movies. This section 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" beginning on page 2-257.

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 (described on page 2-105) 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 (described on page 2-257).

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. (For details on the movie controller component, see *Inside Macintosh: QuickTime Components*.) If you are calling the Movie Toolbox directly to do editing, you should use the functions described in this section.

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.

PutMovieOnScrap

The PutMovieOnScrap function allows your application to place a movie onto the scrap.

```
pascal OSErr PutMovieOnScrap (Movie theMovie,
                              long movieScrapFlags);
```

theMovie Specifies the movie for this operation. Your application obtains this movie identifier from such functions as NewMovie, NewMovieFromFile, and NewMovieFromHandle (described on page 2-92, page 2-88, and page 2-90, respectively).

movieScrapFlags
Flags that control the operation. The following flags are available (set unused flags to 0):

movieScrapDontZeroScrap
Controls whether the Movie Toolbox clears the scrap before putting the movie on the scrap. If you set this flag to 1, the Movie Toolbox does not clear the scrap before placing your movie onto this scrap, thus adding your movie to the previous contents of the scrap. If you set this flag to 0, the function clears the scrap, then places your movie on the scrap.

movieScrapOnlyPutMovie
Controls whether the Movie Toolbox places other items on the scrap along with your movie. If you set this flag to 1, the Movie Toolbox only places your movie on the scrap. If you set this flag to 0, the Movie Toolbox places an image from the current movie time (including but not limited to a PICT) on the scrap along with your movie. The picture is intended for use by applications that cannot work with movies.

ERROR CODES

invalidMovie	–2010	This movie is corrupted or invalid

Image Compression Manager errors
Memory Manager errors

NewMovieFromScrap

The `NewMovieFromScrap` function allows your application to create a movie from the contents of the scrap, if this is possible. If there is no movie data on the scrap, the Movie Toolbox does not create a new movie.

```
pascal Movie NewMovieFromScrap (long newMovieFlags);
```

`newMovieFlags`
Controls the operation of the `NewMovieFromScrap` function. The following flags are available (set unused flags to 0):

`newMovieActive`
Controls whether the new movie is active. Set this flag to 1 to make the new movie active. A movie that does not have any tracks can still be active. When the Movie Toolbox tries to play the movie, no images are displayed, because there is no movie data. Unless you set this flag, you should call the `SetMovieActive` function (described on page 2-145) to play a movie.

`newMovieDontResolveDataRefs`
Controls how completely the Movie Toolbox resolves data references in the movie resource. If you set this flag to 0, the toolbox tries to completely resolve all data references in the resource. This may involve searching for files on remote volumes. If you set this flag to 1, the Movie Toolbox only looks in the specified file.

If the Movie Toolbox cannot completely resolve all the data references, it still returns a valid movie identifier. In this case, the Movie Toolbox also sets the current error value to `couldNotResolveDataRef`.

`newMovieDontAskUnresolvedDataRefs`
Controls whether the Movie Toolbox asks the user to locate files. If you set this flag to 0, the Movie Toolbox asks the user to locate files that it cannot find on available volumes. If the Movie Toolbox cannot locate a file even with the user's help, the function returns a valid movie identifier and sets the current error value to `couldNotResolveDataRef`.

`newMovieDontAutoAlternate`
Controls whether the Movie Toolbox automatically selects enabled tracks from alternate track groups. If you set this flag to 1, the Movie Toolbox does not automatically select tracks for the movie—you must enable tracks yourself.

DESCRIPTION

The `NewMovieFromScrap` function returns the new movie's identifier. If the function cannot load the movie, the returned identifier is set to `nil`.

ERROR CODES

`couldNotResolveDataRef`	–2000	Cannot use this data reference
`cantFindHandler`	–2003	Cannot locate a handler
`cantOpenHandler`	–2004	Cannot open a handler
`invalidMedia`	–2008	This media is corrupted or invalid

File Manager errors
Memory Manager errors

SetMovieSelection

The `SetMovieSelection` function sets a movie's current selection.

```
pascal void SetMovieSelection (Movie theMovie,
                               TimeValue selectionTime,
                               TimeValue selectionDuration);
```

`theMovie` Specifies the movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

`selectionTime`
Contains a time value specifying the starting point of the current selection.

`selectionDuration`
Contains a time value that specifies the duration of the current selection.

DESCRIPTION

If you set the `selectionDuration` parameter to a value greater than the movie's duration, `SetMovieSelection` automatically adjusts the duration of the selection to correspond to the difference between the value specified in the `selectionTime` parameter and the end of the movie.

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid
`invalidTime`	–2015	This time value is invalid

SEE ALSO

You can use the `GetMovieSelection` function, described in the next section, to obtain information about a movie's current selection.

GetMovieSelection

The `GetMovieSelection` function returns information about a movie's current selection.

```
pascal void GetMovieSelection (Movie theMovie,
                               TimeValue *selectionTime,
                               TimeValue *selectionDuration);
```

`theMovie` Specifies the movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

`selectionTime`
Contains a pointer to a time value. The `GetMovieSelection` function places the starting time of the current selection into the field referred to by this parameter. Set this parameter to `nil` if you do not want this information.

`selectionDuration`
Contains a pointer to a time value. The `GetMovieSelection` function places the duration of the current selection into the field referred to by this parameter. Set this parameter to `nil` if you do not want this information.

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid

SEE ALSO

Your application can set the current selection by calling the `SetMovieSelection` function, which is described in the previous section.

CutMovieSelection

The `CutMovieSelection` function creates a new movie that contains the original movie's current selection. This function then removes the current selection from the original movie. After the current selection has been removed from the original movie, the duration of the current selection is 0. The starting time of the current selection is not affected.

```
pascal Movie CutMovieSelection (Movie theMovie);
```

theMovie Specifies the movie for this operation. Your application obtains this movie identifier from such functions as NewMovie, NewMovieFromFile, and NewMovieFromHandle (described on page 2-92, page 2-88, and page 2-90, respectively).

DESCRIPTION

The CutMovieSelection function returns a movie identifier. If the function could not create the new movie, it sets this returned identifier to nil.

Your application must dispose of the new movie once you are done with it. You can use the DisposeMovie function (described on page 2-96) to dispose of the new movie.

If you have assigned a progress function to the source movie, the Movie Toolbox calls that progress function during long cut operations. (For details on progress functions, see "Progress Functions" beginning on page 2-354.)

ERROR CODES

invalidMovie	–2010	This movie is corrupted or invalid
progressProcAborted	–2019	Your progress function returned an error

Memory Manager errors

CopyMovieSelection

The CopyMovieSelection function creates a new movie that contains the original movie's current selection. This function does not change the original movie or the current selection.

```
pascal Movie CopyMovieSelection (Movie theMovie);
```

theMovie Specifies the movie for this operation. Your application obtains this movie identifier from such functions as NewMovie, NewMovieFromFile, and NewMovieFromHandle (described on page 2-92, page 2-88, and page 2-90, respectively).

DESCRIPTION

The CopyMovieSelection function returns a movie identifier. If the function could not create the new movie, it sets this returned identifier to nil.

Your application must dispose of the new movie once you are done with it. You can use the DisposeMovie function (described on page 2-96) to dispose of the new movie.

If you have assigned a progress function to the source movie, the Movie Toolbox calls that progress function during long copy operations. (For details on progress functions, see "Progress Functions" beginning on page 2-354.)

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid
`progressProcAborted`	–2019	Your progress function returned an error

Memory Manager errors

PasteMovieSelection

The `PasteMovieSelection` function places the tracks from one movie into another movie.

```
pascal void PasteMovieSelection (Movie theMovie, Movie src);
```

`theMovie` Specifies the destination movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

`src` Specifies the source movie for this operation. The `PasteMovieSelection` function places the tracks from this movie in the destination movie.

DESCRIPTION

All of the tracks from the source movie are placed in the destination movie. If the duration of the destination movie's current selection is 0, the source movie is inserted at the starting time of the current selection. If the current selection duration is nonzero, the function clears the current selection and then inserts the tracks from the source movie. After the paste operation, the current selection time is unchanged, and the selection duration is set to the source movie's duration.

Whenever possible, the Movie Toolbox uses existing tracks to store the data to be pasted. Before adding a track to the destination movie, the toolbox looks in the destination movie for tracks that have the same characteristics as the tracks in the source movie. The toolbox considers the following characteristics when searching for an appropriate track:

- track spatial dimensions
- track matrix
- track clipping region
- track matte
- alternate group affiliation
- media time scale
- media type
- media language
- data reference (that is, the two tracks must refer to the same file)

If the Movie Toolbox cannot find an appropriate track in the destination movie, it creates a track with the proper characteristics.

The Movie Toolbox removes any empty tracks from the destination movie after the paste operation.

If you have assigned a progress function to the destination movie, the Movie Toolbox calls that progress function during long paste operations. (For details on progress functions, see "Progress Functions" beginning on page 2-354.)

SPECIAL CONSIDERATIONS

The entire source movie is used regardless of the selection in the source movie.

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid
`progressProcAborted`	–2019	Your progress function returned an error

Memory Manager errors

SEE ALSO

If you want to insert only a part of the source movie, use the `InsertMovieSegment` function, which is described on page 2-257.

AddMovieSelection

The `AddMovieSelection` function adds one or more tracks to a movie. This function scales the source movie so that it fits into the destination selection. If the current selection in the destination movie has a 0 duration, the Movie Toolbox adds the segment at the beginning of the current selection.

```
pascal void AddMovieSelection (Movie theMovie, Movie src);
```

`theMovie`	Specifies the destination movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).
`src`	Specifies the source movie for this operation. The `AddMovieSelection` function adds the tracks from this movie to the destination movie. The function adds these tracks at the time specified by the current selection in the destination movie.

DESCRIPTION

The `AddMovieSelection` function is similar to `PasteMovieSelection`, which is described in the previous section. However, the `PasteMovieSelection` function inserts empty space into a movie's existing tracks and then adds the new track data. The `AddMovieSelection` function does not insert empty space into the existing tracks. This function simply adds the tracks in parallel from the source movie to the destination movie. This can be useful for adding a track to an existing movie, such as adding sound to a silent movie.

The Movie Toolbox removes any empty tracks from the destination movie after the add operation.

If you have assigned a progress function to the destination movie, the Movie Toolbox calls that progress function during long add operations. (For details, see "Progress Functions" beginning on page 2-354.)

The entire source movie is used regardless of the selection in the source movie.

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid
`progressProcAborted`	–2019	Your progress function returned an error

Memory Manager errors

ClearMovieSelection

The `ClearMovieSelection` function removes the segment of the movie that is defined by the current selection.

```
pascal void ClearMovieSelection (Movie theMovie);
```

`theMovie` Specifies the movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

DESCRIPTION

After removing the segment, the Movie Toolbox sets the duration of the movie's current selection to 0 and the selection time remains unchanged. This function removes empty tracks from the resulting movie.

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid

IsScrapMovie

The `IsScrapMovie` function looks on the system scrap to find out if it can translate any of the data into a movie.

```
pascal Component IsScrapMovie (Track targetTrack);
```

`targetTrack`
Specifies the location of the potential target movie for the data on the system scrap.

DESCRIPTION

If `IsScrapMovie` finds an appropriate type, it returns a movie import component that can translate the scrap. Otherwise, it returns 0. For details on movie import components, see *Inside Macintosh: QuickTime Components*.

PasteHandleIntoMovie

The `PasteHandleIntoMovie` function takes the contents of a specified handle, together with its type, and pastes it into a specified movie.

```
pascal OSErr PasteHandleIntoMovie (Handle h, OSType handleType,
                                   Movie theMovie, long flags,
                                   ComponentInstance userComp);
```

`h` Specifies the handle to be pasted into the movie indicated by the `handleType` parameter.

`handleType`
Indicates the data type of the handle specified in the `h` parameter.

`theMovie` Specifies the destination movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

`flags` Specifies a constant that further refines conditions of the paste operation.

`pasteInParallel`
Changes the function so that it takes the contents of the specified handle along with its type and adds (rather than inserts) it to the specified movie in an operation analogous to that of the `AddMovieSelection` function. This operation does not affect the duration of existing tracks. It does not necessarily create a new track; rather, it uses a piece of an existing track, if possible.

userComp Specifies the component or an instance of the component that is to perform the conversion of the data into a QuickTime movie. If you want a particular movie import component to perform the conversion, you may pass the component or an instance of that component. Otherwise set this parameter to 0 to allow the Movie Toolbox to determine the appropriate component. If you pass in a component instance, it will be used by PasteHandleIntoMovie. This allows you to communicate directly with the component before using this function to establish any conversion parameters. If you pass in a component ID, an instance is created and closed within this function.

DESCRIPTION

If the handle is set to 0, PasteHandleIntoMovie searches the scrap for a field of the type handleType. If both the h parameter and the handleType parameter are nil, PasteHandleIntoMovie uses the first available data from the scrap.

If you are just pasting in data from the scrap, it is best to allow PasteHandleIntoMovie to retrieve the data from the scrap, rather than doing it yourself. In this way, the function is able to obtain supplemental data from the scrap, if necessary (for example, 'styl' resources for 'TEXT').

PasteHandleIntoMovie pastes into the current selection according to the following rules:

- If the selection is empty (for example, duration = 0), PasteHandleIntoMovie adds the data with the appropriate duration.
- If the selection is not empty, the data is added and then scaled to fit into the duration of the selection. The current selection is deleted, unless you set the pasteInParallel flag.

PutMovieIntoTypedHandle

The PutMovieIntoTypedHandle function takes a movie (or a single track from within that movie) and converts it into a handle of a specified type.

```
pascal OSErr PutMovieIntoTypedHandle (Movie theMovie,
                                      Track targetTrack,
                                      OSType handleType,
                                      Handle publicMovie,
                                      TimeValue start,
                                      TimeValue dur,
                                      long flags,
                                      ComponentInstance userComp);
```

`theMovie` Specifies the movie to convert.

`targetTrack`
Specifies the track to convert.

`handleType`
Indicates the type of the new data.

`publicMovie`
Contains the actual handle in which to place the new data.

`start` Specifies the start time of the segment of the movie or track to be converted.

`dur` Specifies the duration of the segment of the movie or track to be converted.

`flags` Indicates condition of the conversion. Set this parameter to 0.

`userComp` Indicates a component or component instance of the movie export component you want to perform the conversion. Otherwise, set this parameter to 0 for the Movie Toolbox to choose the appropriate component. If you pass in a component instance, it will be used by `PutMovieIntoTypedHandle`. This allows you to communicate directly with the component before using this function to establish any conversion parameters. If you pass in a component ID, an instance is created and closed within this function. For details on movie export components, see *Inside Macintosh: QuickTime Components*.

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" beginning on page 2-157 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.

NewMovieEditState

You can create an edit state by calling the `NewMovieEditState` function. This function creates an edit state that contains all the information describing a movie's content, including the current selection, the movie's tracks, and the media data associated with those tracks.

Note
You must dispose of a movie's edit states *before* you dispose of the movie itself. Use the `DisposeMovieEditState` function (described on page 2-256) to dispose of an edit state. u

```
pascal MovieEditState NewMovieEditState (Movie theMovie);
```

`theMovie` Specifies the movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

DESCRIPTION

The `NewMovieEditState` function returns a movie edit state identifier. You can use this identifier with other Movie Toolbox edit state functions, such as `UseMovieEditState` (described in the next section). If this function could not create the edit state, it sets this returned identifier to `nil`.

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid

Memory Manager errors

UseMovieEditState

Your application can use the `UseMovieEditState` function to return a movie to its condition according to an edit state you created previously.

```
pascal OSErr UseMovieEditState (Movie theMovie,
                                MovieEditState toState);
```

`theMovie` Specifies the movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

`toState` Specifies the edit state for this operation. Your application obtains this edit state identifier when you create the edit state by calling the `NewMovieEditState` function (described in the previous section).

DESCRIPTION

The `UseMovieEditState` function uses the information stored in the edit state to update the movie's contents. This may change the contents of some of the movie's tracks, or it may even add tracks to the movie or remove tracks from the movie. Consequently, the movie's time and spatial characteristics, especially the duration, may change as a result of restoring the saved edit state. Your application creates an edit state by calling the `NewMovieEditState` function, which is described in the previous section.

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid
`invalidEditState`	–2023	This edit state is invalid
`nonMatchingEditState`	–2024	This edit state is not valid for this movie
`staleEditState`	–2025	Movie or track has been disposed

DisposeMovieEditState

The `DisposeMovieEditState` function disposes of an edit state. Your application must dispose of any edit states you create.

Note
You must dispose of a movie's edit states *before* you dispose of the movie itself. u

```
pascal OSErr DisposeMovieEditState (MovieEditState state);
```

`state` Specifies the edit state for this operation. Your application obtains this edit state identifier when you create the edit state by calling the `NewMovieEditState` function.

ERROR CODES

`invalidEditState`	–2023	This edit state is invalid
`staleEditState`	–2025	Movie or track has been disposed

SEE ALSO

You create an edit state by calling the `NewMovieEditState` function, which is discussed on page 2-255.

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," which begins on page 2-243, 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.

InsertMovieSegment

The `InsertMovieSegment` function copies part of one movie to another. You specify the starting time and duration of the source segment and the time in the destination movie at which to place the information.

```
pascal OSErr InsertMovieSegment (Movie srcMovie, Movie dstMovie,
                                 TimeValue srcIn,
                                 TimeValue srcDuration,
                                 TimeValue dstIn);
```

`srcMovie` Specifies the source movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively). The `InsertMovieSegment` function obtains the movie segment from the source movie specified in this parameter.

`dstMovie` Specifies the destination movie for this operation. The `InsertMovieSegment` function places a copy of the segment, which is obtained from the source movie, into this destination movie. The `dstIn` parameter specifies where the segment is inserted.

`srcIn` Specifies the start of the segment in the source movie. The `srcDuration` parameter specifies the segment's duration. This time value must be expressed in the source movie's time scale.

`srcDuration`
: Specifies the duration of the segment in the source movie. This time value must be expressed in the source movie's time scale.

`dstIn`
: Contains a time value specifying where the segment is to be inserted. This time value must be expressed in the destination movie's time scale.

DESCRIPTION

The `InsertMovieSegment` function does not change the source movie. However, the duration of the destination movie is extended to accommodate the inserted segment. You can use this function to add a segment beyond the end of the destination movie—the Movie Toolbox inserts empty space as appropriate.

You can use the `InsertMovieSegment` function to copy data within a single movie. If you are not copying data from one location in a movie to a different point in the same movie, the function may create new tracks, as appropriate.

Whenever possible, the Movie Toolbox uses existing tracks to store the data to be inserted. Before adding a track to the destination movie, the toolbox looks in the destination movie for tracks that have the same characteristics as the tracks in the source movie. The toolbox considers the following characteristics when searching for an appropriate track:

- track spatial dimensions
- track matrix
- track clipping region
- track matte
- alternate group affiliation
- media time scale
- media type
- media language
- data reference (that is, the two tracks must refer to the same file)

If the Movie Toolbox cannot find an appropriate track in the destination movie, it creates a track with the proper characteristics.

If you have assigned a progress function to the destination movie, the Movie Toolbox calls that progress function during long copy operations. For details on application-defined progress functions, see "Progress Functions" beginning on page 2-354.

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid
`invalidDuration`	–2014	This duration value is invalid
`invalidTime`	–2015	This time value is invalid
`progressProcAborted`	–2019	Your progress function returned an error

Memory Manager errors

InsertEmptyMovieSegment

The `InsertEmptyMovieSegment` function adds an empty segment to a movie. You specify the starting time and duration of the empty segment to be added. These times must be expressed in the movie's time scale.

```
pascal OSErr InsertEmptyMovieSegment (Movie dstMovie,
                                      TimeValue dstIn,
                                      TimeValue dstDuration);
```

`dstMovie` Specifies the movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

`dstIn` Contains a time value specifying where the segment is to be inserted. This time value must be expressed in the movie's time scale.

`dstDuration`
Contains a time value that specifies the duration of the segment to be added.

DESCRIPTION

The `InsertEmptyMovieSegment` function then inserts the appropriate amount of empty time into each of the movie's tracks. The exact meaning of the term *empty time* depends upon the type of track. For example, empty time in a sound track is silent.

You cannot add empty space to the end of a movie. If you want to insert a segment beyond the end of a movie, use the `InsertMovieSegment` function, which is described in the previous section.

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid
`invalidDuration`	–2014	This duration value is invalid
`invalidTime`	–2015	This time value is invalid

Memory Manager errors

DeleteMovieSegment

The `DeleteMovieSegment` function removes a specified segment from a movie. You identify the segment to remove by specifying its starting time and duration.

```
pascal OSErr DeleteMovieSegment (Movie theMovie, TimeValue in,
                                 TimeValue duration);
```

`theMovie` Specifies the movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

`in` Contains a time value specifying the starting point of the segment to be deleted.

`duration` Contains a time value that specifies the duration of the segment to be deleted.

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid
`invalidDuration`	–2014	This duration value is invalid
`invalidTime`	–2015	This time value is invalid

ScaleMovieSegment

The `ScaleMovieSegment` function changes the duration of a segment of a movie. The Movie Toolbox scales the segment to accommodate the new duration.

```
pascal OSErr ScaleMovieSegment (Movie theMovie, TimeValue in,
                                TimeValue oldDuration,
                                TimeValue newDuration);
```

`theMovie` Specifies the movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

`in` Specifies the start of the segment. The `oldDuration` parameter specifies the segment's duration. This time value must be expressed in the movie's time scale.

`oldDuration` Specifies the duration of the segment in the source movie. This time value must be expressed in the movie's time scale.

newDuration
: Specifies the new duration of the segment. This time value must be expressed in the movie's time scale. The function alters the segment to accommodate the new duration.

ERROR CODES

invalidMovie	–2010	This movie is corrupted or invalid
invalidDuration	–2014	This duration value is invalid
invalidTime	–2015	This time value is invalid

Memory Manager errors

CopyMovieSettings

The CopyMovieSettings function copies many settings from one movie to another, overwriting the destination settings in the process.

```
pascal OSErr CopyMovieSettings (Movie srcMovie, Movie dstMovie);
```

srcMovie
: Specifies the source movie for this operation. Your application obtains this movie identifier from such functions as NewMovie, NewMovieFromFile, and NewMovieFromHandle (described on page 2-92, page 2-88, and page 2-90, respectively).

dstMovie
: Specifies the destination movie for this operation. The CopyMovieSettings function uses the settings from the source movie, which is specified by the srcMovie parameter, to replace the current settings of this movie.

DESCRIPTION

The CopyMovieSettings function copies the

- preferred rate and volume
- source clipping region
- matrix information
- user data

If you want to work with specific characteristics, you can use the Movie Toolbox functions that allow you to manipulate movie settings individually. These functions are described in "Functions That Modify Movie Properties" beginning on page 2-157.

This function does not copy the movie's contents. To work with movie contents, you should use the segment editing functions described in "Low-Level Movie-Editing Functions" beginning on page 2-257.

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid

Memory Manager errors

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. 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.

InsertTrackSegment

The `InsertTrackSegment` function copies part of one track to another. You specify the starting time and duration of the source segment and the time in the destination track at which to place the information.

```
pascal OSErr InsertTrackSegment (Track srcTrack, Track dstTrack,
                                 TimeValue srcIn,
                                 TimeValue srcDuration,
                                 TimeValue dstIn);
```

`srcTrack` Specifies the source track for this operation. Your application obtains this track identifier from such Movie Toolbox functions as `NewMovieTrack` and `GetMovieTrack` (described on page 2-151 and page 2-204, respectively).

`dstTrack` Specifies the destination track for this operation. The `InsertTrackSegment` function places a copy of the segment, which is obtained from the source track, into this destination track. The `in` parameter specifies where the segment is inserted.

`srcIn` Specifies the start of the segment in the source track. The `srcDuration` parameter specifies the segment's duration. This time value must be expressed in the time scale of the movie that contains the source track.

`srcDuration` Specifies the duration of the segment in the source track. This time value must be expressed in the time scale of the movie that contains the source track.

`dstIn` Contains a time value specifying where the segment is to be inserted. This time value must be expressed in the time scale of the movie that contains the destination track.

DESCRIPTION

The `InsertTrackSegment` function does not change the source track. However, the duration of the destination track is extended to accommodate the inserted segment. This may also change the duration of the movie that contains the destination track.

You can use this function to copy data within a single track. If you are not copying data from one location in a track to a different point in the same track, make sure that the two tracks are of the same type. For example, you cannot copy a segment from a sound track into a video track.

In addition, if the source and destination tracks are associated with different media data files, this function copies samples from the source to the destination using the `AddMediaSample` function. Therefore, the Movie Toolbox must be able to write to the destination media. In this case, your application must call the `BeginMediaEdits` function before calling `InsertTrackSegment`. At the end of the editing session, your application must call the `EndMediaEdits` function. See "Adding Samples to Media Structures" beginning on page 2-271 for more information about these functions.

If you have assigned a progress function to the movie that contains the destination track, the Movie Toolbox calls that progress function during long copy operations.

ERROR CODES

`invalidTrack`	–2009	This track is corrupted or invalid
`mediaTypesDontMatch`	–2018	These media structures don't match
`progressProcAborted`	–2019	Your progress function returned an error

File Manager errors

InsertEmptyTrackSegment

The `InsertEmptyTrackSegment` function adds an empty segment to a track. You specify the starting time and duration of the empty segment to be added. These times must be expressed in the movie's time scale. This function then inserts the appropriate amount of empty time into the track. The exact meaning of the term *empty time* depends upon the type of track. For example, empty time in a sound track is silent.

```
pascal OSErr InsertEmptyTrackSegment (Track dstTrack,
                                      TimeValue dstIn,
                                      TimeValue dstDuration);
```

`dstTrack` Specifies the track for this operation. Your application obtains this track identifier from such Movie Toolbox functions as `NewMovieTrack` and `GetMovieTrack` (described on page 2-151 and page 2-204, respectively).

`dstIn` Contains a time value specifying where the segment is to be inserted. This time value must be expressed in the time scale of the movie that contains the destination track.

`dstDuration`
Contains a time value that specifies the duration of the segment to be added. This time value must be expressed in the time scale of the movie that contains the destination track.

DESCRIPTION

Note that you cannot add empty space to the end of a movie or to the end of a track. If you try to add an empty segment beyond the end of a track, this function does not add the empty segment and returns a result code of `invalidTime`.

ERROR CODES

`invalidTrack`	–2009	This track is corrupted or invalid
`invalidDuration`	–2014	This duration value is invalid
`invalidTime`	–2015	This time value is invalid

Memory Manager errors

InsertMediaIntoTrack

The `InsertMediaIntoTrack` function inserts a reference to a media segment into a track. You specify the segment in the media by providing a starting time and duration. You specify the point in the destination track by providing a time in the track.

```
pascal OSErr InsertMediaIntoTrack (Track theTrack,
                                   TimeValue trackStart,
                                   TimeValue mediaTime,
                                   TimeValue mediaDuration,
                                   Fixed mediaRate);
```

`theTrack` Specifies the track for this operation. Your application obtains this track identifier from such Movie Toolbox functions as `NewMovieTrack` and `GetMovieTrack` (described on page 2-151 and page 2-204, respectively).

`trackStart`
Contains a time value specifying where the segment is to be inserted. This time value must be expressed in the movie's time scale. If you set this parameter to –1, the media data is added to the end of the track.

`mediaTime` Contains a time value specifying the starting point of the segment in the media. This time value must be expressed in the media's time scale.

`mediaDuration`
Contains a time value specifying the duration of the media's segment. This time value must be expressed in the media's time scale.

`mediaRate` Specifies the media's rate. A value of 1.0 indicates the media's natural playback rate. This value should be a positive, nonzero rate.

DESCRIPTION

The `InsertMediaIntoTrack` function inserts the media segment into the track at the specified location. The Movie Toolbox determines the duration of the segment in the track based on the media rate and duration information you provide.

You use this function after you have added samples to a media using the functions described in "Adding Samples to Media Structures" beginning on page 2-271. If you play the track before you call this function, the track does not contain the new media data.

ERROR CODES

`invalidTrack`	–2009	This track is corrupted or invalid
`invalidDuration`	–2014	This duration value is invalid
`invalidTime`	–2015	This time value is invalid

DeleteTrackSegment

The `DeleteTrackSegment` function removes a specified segment from a track. You identify the segment to remove by specifying its starting time and duration.

```
pascal OSErr DeleteTrackSegment (Track theTrack, TimeValue in,
                                 TimeValue duration);
```

`theTrack` Specifies the track for this operation. Your application obtains this track identifier from such Movie Toolbox functions as `NewMovieTrack` and `GetMovieTrack` (described on page 2-151 and page 2-204, respectively).

`in` Contains a time value specifying the starting point of the segment to be deleted. This time value must be expressed in the time scale of the movie that contains the source track.

`duration` Contains a time value that specifies the duration of the segment to be deleted. This time value must be expressed in the time scale of the movie that contains the source track.

ERROR CODES

`invalidTrack`	–2009	This track is corrupted or invalid
`invalidDuration`	–2014	This duration value is invalid
`invalidTime`	–2015	This time value is invalid

SEE ALSO

To dispose of a track, call the `DisposeMovieTrack` function, described on page 2-152.

ScaleTrackSegment

The `ScaleTrackSegment` function changes the duration of a segment of a track. This may change the duration of the movie that contains the track. However, this function does not cause the Movie Toolbox to add data to or remove data from the movie.

```
pascal OSErr ScaleTrackSegment (Track theTrack, TimeValue in,
                                TimeValue oldDuration,
                                TimeValue newDuration);
```

`theTrack` Specifies the track for this operation. Your application obtains this track identifier from such Movie Toolbox functions as `NewMovieTrack` and `GetMovieTrack` (described on page 2-151 and page 2-204, respectively).

`in` Specifies the start of the segment. The `oldDuration` parameter specifies the segment's duration. This time value must be expressed in the time scale of the movie that contains the track.

`oldDuration`
Specifies the duration of the segment. This time value must be expressed in the time scale of the movie that contains the track.

`newDuration`
Specifies the new duration of the segment. This time value must be expressed in the time scale of the movie that contains the track. The function alters the segment to accommodate the new duration.

ERROR CODES

`invalidTrack`	–2009	This track is corrupted or invalid
`invalidDuration`	–2014	This duration value is invalid
`invalidTime`	–2015	This time value is invalid

Memory Manager errors

CopyTrackSettings

The `CopyTrackSettings` function copies many settings from one track to another, overwriting the destination settings.

```
pascal OSErr CopyTrackSettings (Track srcTrack, Track dstTrack);
```

`srcTrack` Specifies the source track for this operation. Your application obtains this track identifier from such Movie Toolbox functions as `NewMovieTrack` and `GetMovieTrack` (described on page 2-151 and page 2-204, respectively).

`dstTrack` Specifies the destination track for this operation. The `CopyTrackSettings` function uses the settings from the source track, which you specify with the `srcTrack` parameter, to replace the current settings of this track.

DESCRIPTION

The `CopyTrackSettings` function copies the

- matrix information
- track volume
- clipping region
- user data
- matte information
- media language, quality, and user data
- other media-specific settings (such as sound balance and video graphics mode)

This function does not copy any alternate group information pertaining to the track.

If you want to work with specific characteristics, you can use the Movie Toolbox functions that allow you to manipulate track settings individually. These functions are described in "Functions That Modify Movie Properties," which begins on page 2-157.

This function does not copy the track's contents. To work with track contents, you should use the segment-editing functions described in "Editing Tracks" beginning on page 2-262.

ERROR CODES

`invalidTrack`	–2009	This track is corrupted or invalid

Memory Manager errors

GetTrackEditRate

The `GetTrackEditRate` function returns the rate of the track edit of a specified track at an indicated time.

```
pascal Fixed GetTrackEditRate (Track theTrack, TimeValue atTime);
```

`theTrack`	Specifies the track identifier for which the rate of a track edit (at the time given in the `atTime` parameter) is to be determined.
`atTime`	Indicates a time value at which the rate of a track edit (of a track identified in the parameter `theTrack`) is to be determined.

DESCRIPTION

If an invalid time or track is passed, the returned value is 0.0. The track edit rate is typically 1.0, unless either the `ScaleMovieSegment` or `ScaleTrackSegment` function has been called. (For more on the `ScaleMovieSegment` and `ScaleTrackSegment` functions, see page 2-260 and page 2-266, respectively.)

The `GetTrackEditRate` function is relevant if you are stepping through track edits directly in your application or if you are a client of the **base media handler.** (See *Inside Macintosh: QuickTime Components* for details on media handlers.)

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," which begins on page 2-157, 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.

NewTrackEditState

You can create an edit state by calling the `NewTrackEditState` function. This function creates an edit state that contains all the information describing a track's content, including the identity of the media data associated with the track and all the track's edit lists.

Note
You must dispose of a movie's track edit states *before* disposing of the track or of the movie that contains the track. Use the `DisposeTrackEditState` function, which is described on page 2-270, to dispose of an edit state. u

```
pascal TrackEditState NewTrackEditState (Track theTrack);
```

`theTrack` Specifies the track for this operation. Your application obtains this track identifier from such Movie Toolbox functions as `NewMovieTrack` and `GetMovieTrack` (described on page 2-151 and page 2-204, respectively).

DESCRIPTION

The `NewTrackEditState` function returns a track edit state identifier. You can use this identifier with other Movie Toolbox edit state functions, such as `UseTrackEditState` (described in the next section). If this function could not create the edit state, it sets this returned identifier to `nil`.

ERROR CODES

`invalidTrack`	–2009	This track is corrupted or invalid

Memory Manager errors

UseTrackEditState

Your application can use the `UseTrackEditState` function to return a track to its condition according to an edit state you created previously.

```
pascal OSErr UseTrackEditState (Track theTrack,
                                TrackEditState state);
```

`theTrack` Specifies the track for this operation. Your application obtains this track identifier from such Movie Toolbox functions as `NewMovieTrack` and `GetMovieTrack` (described on page 2-151 and page 2-204, respectively).

`state` Specifies the edit state for this operation. Your application obtains this edit state identifier when you create the edit state by calling the `NewTrackEditState` function, which is described in the previous section.

DESCRIPTION

The `UseTrackEditState` function uses the information stored in the edit state to update the track's contents. This may change the contents of some of the track. Consequently, the time characteristics of the movie that contains the track, especially the duration, may change as a result of restoring the saved edit state. Your application creates an edit state by calling the `NewTrackEditState` function.

SPECIAL CONSIDERATIONS

You can use the `UseTrackEditState` function only with tracks that currently belong to a movie. A track may be detached from its movie as a result of edit processing—you cannot use this function with such a track.

ERROR CODES

`invalidTrack`	–2009	This track is corrupted or invalid
`invalidEditState`	–2023	This edit state is invalid
`nonMatchingEditState`	–2024	This edit state is not valid for this movie

DisposeTrackEditState

The `DisposeTrackEditState` function disposes of a track edit state. Your application must dispose of any edit states you create. You create an edit state by calling the `NewTrackEditState` function, which is discussed on page 2-269.

Note

You must dispose of a movie's track edit states *before* you dispose of the track or the movie. u

```
pascal OSErr DisposeTrackEditState (TrackEditState state);
```

`state` Specifies the edit state for this operation. Your application obtains this edit state identifier when you create the edit state by calling the `NewTrackEditState` function (described on page 2-269).

ERROR CODES

`invalidTrack`	–2009	This track is corrupted or invalid
`invalidEditState`	–2023	This edit state is invalid
`staleEditState`	–2025	Movie or track has been disposed

Adding Samples to Media Structures

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*. For more information on key frames, see the chapter "Image Compression Manager" in this book.

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.

BeginMediaEdits

The `BeginMediaEdits` function starts a media-editing session.

```
pascal OSErr BeginMediaEdits (Media theMedia);
```

`theMedia` Specifies the media for this operation. Your application obtains this media identifier from such Movie Toolbox functions as `NewTrackMedia` and `GetTrackMedia` (described on page 2-153 and page 2-206, respectively).

DESCRIPTION

You use the `BeginMediaEdits` function to notify the Movie Toolbox that you are going to add sample data to a media. In response, the Movie Toolbox determines whether the media can be updated. For example, if the media data are stored on disk, the Movie Toolbox opens the disk file with write permissions. If the media is stored on a read-only storage medium, such as a CD-ROM disc, the Movie Toolbox does not start an editing session and returns an error.

Use the `EndMediaEdits` function, which is described in the next section, to end a media-editing session.

You must call `BeginMediaEdits` before you add samples to a media with the `AddMediaSample` function (described on page 2-273). Under some circumstances, you must start a media-editing session before calling the `InsertTrackSegment` function (described on page 2-262).

ERROR CODES

`invalidMedia`	–2008	This media is corrupted or invalid

File system errors

EndMediaEdits

The `EndMediaEdits` function ends a media-editing session.

```
pascal OSErr EndMediaEdits (Media theMedia);
```

`theMedia` Specifies the media for this operation. Your application obtains this media identifier from such Movie Toolbox functions as `NewTrackMedia` and `GetTrackMedia` (described on page 2-153 and page 2-206, respectively).

DESCRIPTION

You use the `EndMediaEdits` function to tell the Movie Toolbox that you are done adding samples to a movie data file. The Movie Toolbox then performs the appropriate processing. For example, for disk-based media, the Movie Toolbox relinquishes write-access to the disk file. You should call `EndMediaEdits` only if you successfully started a media-editing session with the `BeginMediaEdits` function, which is described in the previous section.

ERROR CODES

`invalidMedia`	–2008	This media is corrupted or invalid

AddMediaSample

The `AddMediaSample` function adds sample data and a description to a media. Your application specifies the sample and the media for the operation. The `AddMediaSample` function updates the media so that it contains the sample data. One call to this function can add several samples to a media—however, all the samples must be the same size. Samples are always appended to the end of the media. Furthermore, each time a sample is added, the media duration is extended.

```
pascal OSErr AddMediaSample (Media theMedia, Handle dataIn,
                    long inOffset, unsigned long size,
                    TimeValue durationPerSample,
                    SampleDescriptionHandle sampleDescriptionH,
                    long numberOfSamples, short sampleFlags,
                    TimeValue *sampleTime);
```

`theMedia` Specifies the media for this operation. Your application obtains this media identifier from such Movie Toolbox functions as `NewTrackMedia` and `GetTrackMedia` (described on page 2-153 and page 2-206, respectively).

`dataIn` Contains a handle to the sample data. The `AddMediaSample` function adds this data to the media specified by the parameter `theMedia`. You specify the number of bytes of sample data with the `size` parameter. You can use the `inOffset` parameter to specify a byte offset into the data referred to by this handle.

`inOffset` Specifies an offset into the data referred to by the handle contained in the `dataIn` parameter. Set this parameter to 0 if there is no offset.

`size` Specifies the number of bytes of sample data to be added to the media. This parameter indicates the total number of bytes in the sample data to be added to the media, not the number of bytes per sample. Use the `numberOfSamples` parameter to indicate the number of samples that are contained in the sample data.

`durationPerSample`
Specifies the duration of each sample to be added. You must specify this parameter in the media's time scale. For example, if you are adding sound that was sampled at 22 kHz to a media that contains a sound track with the same time scale, you would set the `durationPerSample` parameter to 1. Similarly, if you are adding video that was recorded at 10 frames per second to a video media that has a time scale of 600, you would set this parameter to 60 to add a single sample.

`sampleDescriptionH`
Contains a handle to a sample description. Some media structures may require sample descriptions. There are different sample descriptions for different types of samples. For example, a media that contains compressed video requires that you supply an image description (see the chapter "Image Compression Manager" in this book for more information about image description structures). A media that contains sound requires

that you supply a sound description structure (see "The Sound Description Structure" on page 2-79 for more information about sound description structures).

If the media does not require a sample description, set this parameter to `nil`.

`numberOfSamples`

Specifies the number of samples contained in the sample data to be added to the media.

This parameter determines the size of each sample. The Movie Toolbox considers the value of this parameter as well as the value of the `size` parameter when it determines the size of each sample that it adds to the media. You should set the value of this parameter so that the resulting sample size represents a reasonable compromise between total data retrieval time and the overhead associated with input and output (I/O). You should also consider the speed of the data storage device—CD-ROM devices are much slower than hard disks, for example, and should therefore have a smaller sample size.

For a video media, set a sample size that corresponds to the size of a frame. For a sound media, choose a number of samples that corresponds to between 0.5 and 1.0 seconds of sound. In general, you should not create groups of sound samples that are less than 2 KB in size or greater than 15 KB. Typically, a sample size of about 8 KB is reasonable for most storage devices.

`sampleFlags`

Contains flags that control the add operation. The following flag is available (set unused flags to 0):

`mediaSampleNotSync`

Indicates that the sample to be added is not a sync sample. Set this flag to 1 if the sample is not a sync sample. Set this flag to 0 if the sample is a sync sample.

`sampleTime`

Contains a pointer to a time value. After adding the sample data to the media, the `AddMediaSample` function returns the time where the sample was inserted in the time value referred to by this parameter. If you do not want to receive this information, set this parameter to `nil`.

DESCRIPTION

The `AddMediaSample` function updates the file or device that contains the movie data file as part of the add operation. Consequently, your application must have started a media-editing session before calling this function. You start a media-editing session with the `BeginMediaEdits` function, which is described on page 2-271. If you want to work with samples that have already been added to a movie data file, use the `AddMediaSampleReference` function, which is described in the next section.

ERROR CODES

`invalidMedia`	–2008	This media is corrupted or invalid

File Manager errors
Memory Manager errors

AddMediaSampleReference

The `AddMediaSampleReference` function allows your application to work with samples that have already been added to a movie data file. Instead of actually writing out samples to disk, this function writes out references to existing samples, which you specify in the `dataOffset` and `size` parameters.

```
pascal OSErr AddMediaSampleReference (Media theMedia,
                    long dataOffset,
                    unsigned long size,
                    TimeValue durationPerSample,
                    SampleDescriptionHandle sampleDescriptionH,
                    long numberOfSamples, short sampleFlags,
                    TimeValue *sampleTime);
```

`theMedia` Specifies the media for this operation. Your application obtains this media identifier from such Movie Toolbox functions as `NewTrackMedia` and `GetTrackMedia` (described on page 2-153 and page 2-206, respectively).

`dataOffset`
Specifies the offset into the movie data file. This parameter is used differently by each data handler. For example, for the standard HFS data handler, this parameter specifies the offset into the file. This parameter contains either data you add yourself or the data offset returned by the `GetMediaSampleReference` function (described on page 2-279).

`size` Specifies the number of bytes of sample data to be identified by the reference. This parameter indicates the total number of bytes in the sample data, not the number of bytes per sample. Use the `numberOfSamples` parameter to indicate the number of samples that are contained in the reference.

`durationPerSample`
Specifies the duration of each sample in the reference. You must specify this parameter in the media's time scale. For example, if you are referring to sound that was sampled at 22 kHz in a media that contains a sound track with the same time scale, to add a reference to a single sample you would set the `durationPerSample` parameter to 1. Similarly, if you are referring to video that was recorded at 10 frames per second in a video media that has a time scale of 60, you would set this parameter to 6 to add a reference to a single sample.

sampleDescriptionH
: Contains a handle to a sample description. Some media structures may require sample descriptions. There are different sample descriptions for different types of samples. For example, a media that contains compressed video requires that you supply an image description (see the chapter "Image Compression Manager" in this book for more information about image description structures). A media that contains sound requires that you supply a sound description structure (see "The Sound Description Structure" on page 2-79 for more information about sound description structures).

  If the media does not require a sample description, set this parameter to `nil`.

numberOfSamples
: Specifies the number of samples contained in the reference. For details, see the `AddMediaSample` function description beginning on page 2-273.

sampleFlags
: Contains flags that control the operation. The following flag is available (set unused flags to 0):

  mediaSampleNotSync
  : Indicates that the sample to be added is not a sync sample. Set this flag to 1 if the sample is not a sync sample. Set this flag to 0 if the sample is a sync sample.

sampleTime
: Contains a pointer to a time value. After adding the reference to the media, the `AddMediaSampleReference` function returns the time where the reference was inserted in the time value referred to by this parameter. If you do not want to receive this information, set this parameter to `nil`.

DESCRIPTION

The `AddMediaSampleReference` function does not add sample data to the file or device that contains a media. Rather, it defines references to sample data that you previously added to a movie data file. As with the `AddMediaSample` function (described in the previous section), your application specifies the media for the operation. Note that one reference may refer to more than one sample—all the samples described by a reference must be the same size. This function does not update the movie data file as part of the add operation. Therefore, your application does not have to call the `BeginMediaEdits` function (described on page 2-271) before calling `AddMediaSampleReference`.

ERROR CODES

`invalidMedia`	–2008	This media is corrupted or invalid

Memory Manager errors

SEE ALSO

If you want to add new samples to a media data file, use the `AddMediaSample` function, which is described in the previous section.

GetMediaSample

The `GetMediaSample` function returns a sample from a movie data file. You add samples to movie data files with the `AddMediaSample` function (described on page 2-273).

```
pascal OSErr GetMediaSample (Media theMedia, Handle dataOut,
                    long maxSizeToGrow, long *size,
                    TimeValue time, TimeValue *sampleTime,
                    TimeValue *durationPerSample,
                    SampleDescriptionHandle sampleDescriptionH,
                    long *sampleDescriptionIndex,
                    long maxNumberOfSamples,
                    long *numberOfSamples,
                    short *sampleFlags);
```

`theMedia` Specifies the media for this operation. Your application obtains this media identifier from such Movie Toolbox functions as `NewTrackMedia` and `GetTrackMedia` (described on page 2-153 and page 2-206, respectively).

`dataOut` Contains a handle. The `GetMediaSample` function returns the sample data into this handle. The function increases the size of this handle, if necessary. You can specify the handle's maximum size with the `maxSizeToGrow` parameter.

`maxSizeToGrow` Specifies the maximum number of bytes of sample data to be returned. The `GetMediaSample` function does not increase the handle specified by the `dataOut` parameter to a size greater than you specify with this parameter. Set this value to 0 to enforce no limit on the number of bytes to be returned.

`size` Contains a pointer to a long integer. The `GetMediaSample` function updates the field referred to by the `size` parameter with the number of bytes of sample data returned in the handle specified by the `dataOut` parameter. Set this parameter to `nil` if you are not interested in this information.

`time` Specifies the starting time of the sample to be retrieved. You must specify this value in the media's time scale.

sampleTime
: Contains a pointer to a time value. The GetMediaSample function updates this time value to indicate the actual time of the returned sample data. If you are not interested in this information, set this parameter to nil.

  The returned time may differ from the time you specified with the time parameter. This will occur if the time you specified falls in the middle of a sample.

durationPerSample
: Contains a pointer to a time value. The Movie Toolbox returns the duration of each sample in the media. This time value is expressed in the media's time scale. Set this parameter to 0 if you do not want this information.

sampleDescriptionH
: Contains a handle to a sample description. The GetMediaSample function returns the sample description corresponding to the returned sample data. The function resizes this handle as appropriate. If you do not want the sample description, set this parameter to nil.

sampleDescriptionIndex
: Contains a pointer to a long integer. The GetMediaSample function returns an index value to the sample description that corresponds to the returned sample data. If you do not want this information, set this parameter to nil.

  You can use this index to retrieve the sample description by calling the GetMediaSampleDescription function, which is described on page 2-226.

  You can retrieve the sample description itself by using the sampleDescriptionH parameter.

maxNumberOfSamples
: Specifies the maximum number of samples to be returned. The Movie Toolbox does not return more samples than you specify with this parameter.

  If you set this parameter to 0, the Movie Toolbox uses a value that is appropriate for the media, and returns that value in the field referenced by the numberOfSamples parameter.

numberOfSamples
: Contains a pointer to a long integer. The GetMediaSample function updates the field referred to by this parameter with the number of samples it actually returns. If you do not want this information, set this parameter to nil.

sampleFlags
Contains a pointer to a short integer. The GetMediaSample function returns flags that describe the sample. The following flag is available (set unused flags to 0):

mediaSampleNotSync
Indicates that the sample that is returned is not a sync sample. Set this flag to 1 if the sample is not a sync sample. Set this flag to 0 if the sample is a sync sample.

If you do not want this information, set this parameter to nil.

ERROR CODES

invalidMedia	–2008	This media is corrupted or invalid

File Manager errors
Memory Manager errors

GetMediaSampleReference

The GetMediaSampleReference function allows your application to obtain reference information about samples that are stored in a movie data file.

```
pascal OSErr GetMediaSampleReference (Media theMedia,
                    long *dataOffset,long *size, TimeValue time,
                    TimeValue *sampleTime,
                    TimeValue *durationPerSample,
                    SampleDescriptionHandle sampleDescriptionH,
                    long *sampleDescriptionIndex,
                    long maxNumberOfSamples,
                    long *numberOfSamples, short *sampleFlags);
```

theMedia Specifies the media for this operation. Your application obtains this media identifier from such Movie Toolbox functions as NewTrackMedia and GetTrackMedia (described on page 2-153 and page 2-206, respectively).

dataOffset
Contains a pointer to a long integer. The GetMediaSampleReference function updates the field referred to by this parameter with the offset to the sample data.

This parameter is used differently by each media handler. For example, the hierarchical file system (HFS) media handler returns an offset into the file that contains the media data.

`size` Contains a pointer to a long integer. The `GetMediaSampleReference` function updates the field referred to by the `size` parameter with the number of bytes of sample data referred to by the reference. Set this parameter to `nil` if you are not interested in this information.

`time` Specifies the starting time of the sample reference to be retrieved. You must specify this value in the media's time scale.

`sampleTime`
Contains a pointer to a time value. The `GetMediaSampleReference` function updates this time value to indicate the actual time of the returned sample data. If you are not interested in this information, set this parameter to `nil`.

The returned time may differ from the time you specified with the `time` parameter. This will occur if the time you specified falls in the middle of a sample.

`durationPerSample`
Contains a pointer to a time value. The Movie Toolbox returns the duration of each sample in the media. This time value is expressed in the media's time scale. Set this parameter to 0 if you do not want this information.

`sampleDescriptionH`
Contains a handle to a sample description. The `GetMediaSampleReference` function returns the sample description corresponding to the returned sample data. The function resizes this handle as appropriate. If you do not want the sample description, set this parameter to `nil`.

`sampleDescriptionIndex`
Contains a pointer to a long integer. The `GetMediaSampleReference` function returns an index value to the sample description that corresponds to the returned sample data. You can use this index to retrieve the media sample description with the `GetMediaSampleDescription` function, which is described on page 2-226. If you do not want this information, set this parameter to `nil`.

You can retrieve the sample description itself by using the `sampleDescriptionH` parameter.

`maxNumberOfSamples`
Specifies the maximum number of samples to be returned. The Movie Toolbox does not return a reference that refers to more samples than you specify with this parameter.

If you set this parameter to 0, the Movie Toolbox uses a value that is appropriate for the media and returns that value in the field referenced by the `numberOfSamples` parameter.

`numberOfSamples`
Contains a pointer to a long integer. The `GetMediaSampleReference` function updates the field referred to by this parameter with the number of samples referred to by the returned reference. If you do not want this information, set this parameter to `nil`.

`sampleFlags`

Contains a pointer to a short integer. The `GetMediaSampleReference` function returns flags that describe the samples referred to by the reference. The following flag is available (unused flags are set to 0):

`mediaSampleNotSync`

Indicates the sample that is returned is not a sync sample. Set this flag to 1 if the sample is not a sync sample. Set this flag to 0 if the sample is a sync sample.

If you do not want this information, set this parameter to `nil`.

DESCRIPTION

The `GetMediaSampleReference` function is similar to `GetMediaSample`, except that it does not return the sample data.

ERROR CODES

`invalidMedia`	–2008	This media is corrupted or invalid

Memory Manager errors

Media Functions

The Movie Toolbox does not contain any support for specific media types. Rather, it delegates this work to media handler components. The Movie Toolbox provides a number of functions that allow your application to interact with media handlers. This section describes those functions. It has been divided into the following topics:

- "Selecting Media Handlers" describes the functions that you can use to gain access to a media handler
- "Video Media Handler Functions" describes the functions that allow your application to interact with video media handlers
- "Sound Media Handler Functions" describes the functions that allow your application to interact with sound media handlers
- "Text Media Handler Functions" describes the functions that allow your application to interact with text media handlers

Selecting Media Handlers

Media handler components are responsible for interpreting and manipulating a media's sample data. Each type of media has its own media handler, which deals with the specific characteristics of the media data. The Movie Toolbox provides a set of functions that allow you to gather information about a media handler and assign a particular media handler to a media. This section discusses those functions.

Each media handler has an associated data handler for each data reference. The data handler is responsible for fetching, storing, and caching the data that the media handler uses. The Movie Toolbox provides functions that allow you to get information about data handlers and to assign a particular data handler to a media.

The `GetMediaHandler` and `GetMediaHandlerDescription` functions allow you to retrieve information about a media handler.

You can use the `SetMediaHandler` function to assign a media handler to a media.

The `GetMediaDataHandler` and `GetMediaDataHandlerDescription` functions enable you to retrieve information about a data handler. Use the `SetMediaDataHandler` function to assign a data handler to a media.

GetMediaHandlerDescription

The `GetMediaHandlerDescription` function allows your application to retrieve information about a media handler. You specify the media.

```
pascal void GetMediaHandlerDescription (Media theMedia,
                                        OSType *mediaType,
                                        Str255 creatorName,
                                        OSType *creatorManufacturer);
```

`theMedia` Specifies the media for this operation. Your application obtains this media identifier from such Movie Toolbox functions as `NewTrackMedia` and `GetTrackMedia` (described on page 2-153 and page 2-206, respectively).

`mediaType` Contains a pointer to a field of data type `OSType`. The Movie Toolbox returns the media type identifier. This value indicates the type of media supported by this media handler. This value also corresponds to the component subtype specified for the media handler component. If you do not want to receive this information, set the `mediaType` parameter to `nil`. The following values are available:

`VideoMediaType`	Video media
`SoundMediaType`	Sound media
`TextMediaType`	Text media

`creatorName`
Points to a string. The Movie Toolbox returns the name of the media handler's creator. If you do not want to receive this information, set this parameter to `nil`.

`creatorManufacturer`
Contains a pointer to a long integer. The Movie Toolbox returns the 4-byte value that identifies the manufacturer of the component. If you do not want to retrieve this information, set this parameter to `nil`.

DESCRIPTION

The Movie Toolbox returns information about that media's media handler. This information describes the media handler that created the media, not the handler that is currently assigned to the media.

ERROR CODES

`invalidMedia`	–2008	This media is corrupted or invalid

GetMediaHandler

The `GetMediaHandler` function allows you to obtain a reference to a media handler component.

You can use this reference to call the media handler directly. See "Video Media Handler Functions," which begins on page 2-287, and "Sound Media Handler Functions," which begins on page 2-288, for information about the functions that are supported by video and sound media handlers.

```
pascal MediaHandler GetMediaHandler (Media theMedia);
```

`theMedia` Specifies the media for this operation. Your application obtains this media identifier from such Movie Toolbox functions as `NewTrackMedia` and `GetTrackMedia` (described on page 2-153 and page 2-206, respectively).

DESCRIPTION

The `GetMediaHandler` function returns a reference to the media's media handler. If the function could not locate the media handler, it sets this reference to `nil`. You can use this reference to call the media handler.

ERROR CODES

`invalidMedia`	–2008	This media is corrupted or invalid

SetMediaHandler

The `SetMediaHandler` function allows you to assign a specific media handler to a track. The Movie Toolbox closes the track's previous media handler and then opens the new one. It is your responsibility to ensure that the media handler you specify can handle the data in the track.

```
pascal OSErr SetMediaHandler (Media theMedia,
                              MediaHandlerComponent mH);
```

`theMedia` Specifies the track for this operation. Your application obtains this track identifier from such Movie Toolbox functions as `NewMovieTrack` and `GetMovieTrack` (described on page 2-151 and page 2-204, respectively).

`mH` Contains a reference to a media handler component. You obtain this reference from the `GetMediaHandler` function, which is described in the previous section.

Note
Your application should not need to call the `SetMediaHandler` function. The Movie Toolbox assigns a media handler to each track when you load a movie. u

ERROR CODES

`invalidHandler`	–2013	This handler is invalid

GetMediaDataHandlerDescription

The `GetMediaDataHandlerDescription` function allows your application to retrieve information about a media's data handler. You specify the media.

```
pascal void GetMediaDataHandlerDescription (Media theMedia,
                                    short index, OSType *dhType,
                                    Str255 creatorName,
                                    OSType *creatorManufacturer);
```

`theMedia` Specifies the media for this operation. Your application obtains this media identifier from such Movie Toolbox functions as `NewTrackMedia` and `GetTrackMedia` (described on page 2-153 and page 2-206, respectively).

`index` Identifies the data reference. You provide the index value that corresponds to the data reference for which you want to retrieve the data handler description. You must set this parameter to 1.

dhType Contains a pointer to a field of data type OSType. The Movie Toolbox returns the data handler type identifier. This value indicates the type of data reference supported by this data handler. This value also corresponds to the component subtype specified for the data handler component. All QuickTime data references have a type value of 'alis'. If you do not want to receive this information, set the dhType parameter to nil.

creatorName
Points to a string. The Movie Toolbox returns the name of the data handler's creator. If you do not want to receive this information, set this parameter to nil.

creatorManufacturer
Contains a pointer to a long integer. The Movie Toolbox returns the 4-byte value that identifies the manufacturer of the component. If you do not want to retrieve this information, set this parameter to nil.

DESCRIPTION

The Movie Toolbox returns information about that media's data handler. This information describes the data handler that created the media data, not the handler that is currently assigned to the media.

ERROR CODES

invalidMedia	–2008	This media is corrupted or invalid

GetMediaDataHandler

The GetMediaDataHandler function allows you to determine a media's data handler.

```
pascal DataHandler GetMediaDataHandler (Media theMedia,
                                        short index);
```

theMedia Specifies the media for this operation. Your application obtains this media identifier from such Movie Toolbox functions as NewTrackMedia and GetTrackMedia (described on page 2-153 and page 2-206, respectively).

index Identifies the data reference. You provide the index value that corresponds to the data reference for which you want to retrieve the data handler. You must set this parameter to 1.

DESCRIPTION

The `GetMediaDataHandler` function returns a data handler identifier. This identifier is a component instance that specifies a connection to a data handler component (see the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for more information about components). If the Movie Toolbox cannot determine the data handler for the media you specify, the function sets this returned value to `nil`.

Note
Your application should not need to call this function. u

ERROR CODES

`invalidMedia`	–2008	This media is corrupted or invalid

SetMediaDataHandler

The `SetMediaDataHandler` function allows you to assign a data handler to a media.

```
pascal OSErr SetMediaDataHandler (Media theMedia, short index,
                                  DataHandlerComponent dataHandler);
```

`theMedia` Specifies the media for this operation. Your application obtains this media identifier from such Movie Toolbox functions as `NewTrackMedia` and `GetTrackMedia` (described on page 2-153 and page 2-206, respectively).

`index` Identifies the data reference for this data handler. You provide the index value that corresponds to the data reference. You must set this parameter to 1.

`dataHandler`
Specifies the data handler for the media. This identifier is a component instance that specifies a connection to a data handler component (see the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for more information about components). If the data handler you specify cannot work with the data stored in the media, the function does not change the media's data handler.

DESCRIPTION

When you create a new media or load an existing media into memory, the media handler assigns an appropriate data handler to the track's media.

Note
Your application should not call the `SetMediaDataHandler` function. The Movie Toolbox assigns a data handler to each media when you load a movie. u

ERROR CODES

`badComponentType`	–2005	Component cannot accommodate this data
`invalidMedia`	–2008	This media is corrupted or invalid

Video Media Handler Functions

Video media handlers are responsible for interpreting and manipulating video data. These media handlers allow you to call them directly to work with some graphics settings. This section describes the functions supported by video media handlers.

Video media handlers maintain a graphics mode and color value that affect the display of video data. You can use the `SetVideoMediaGraphicsMode` and `GetVideoMediaGraphicsMode` functions to work with these characteristics. See *Inside Macintosh: Imaging* for more information about setting color values for use with the `addPin`, `subPin`, `blend`, and `transparent` drawing modes.

Sample descriptions for video media are stored in image description structures. For a complete discussion of the format and content of the image description structure, see the chapter "Image Compression Manager" in this book.

SetVideoMediaGraphicsMode

The `SetVideoMediaGraphicsMode` function allows you to set the graphics mode and blend color of a video media.

```
pascal HandlerError SetVideoMediaGraphicsMode (MediaHandler mh,
                                               long graphicsMode,
                                               const RGBColor *opColor);
```

`mH` Contains a reference to a media handler. You obtain this reference from the `GetMediaHandler` function, which is described on page 2-283.

`graphicsMode`
Specifies the graphics mode of the media handler. This is a QuickDraw transfer mode value.

`opColor` Contains a pointer to the color for use in blending and transparent operations. The media handler passes this color to QuickDraw as appropriate when you draw in `addPin`, `subPin`, `blend`, or `transparent` mode.

ERROR CODES

Component Manager errors

SEE ALSO

You can retrieve the graphics mode and blend color currently in use by a video media handler by calling the `GetVideoMediaGraphicsMode` function, which is described in the next section.

GetVideoMediaGraphicsMode

The `GetVideoMediaGraphicsMode` function allows you to obtain the graphics mode and blend color values currently in use by a video media handler.

```
pascal HandlerError GetVideoMediaGraphicsMode (MediaHandler mh,
                                               long *graphicsMode,
                                               RGBColor *opColor);
```

`mH` Contains a reference to a media handler. You obtain this reference from the `GetMediaHandler` function, which is described on page 2-283.

`graphicsMode`
Contains a pointer to a long integer. The media handler returns the graphics mode currently in use by the media handler. This is a QuickDraw transfer mode value.

`opColor` Contains a pointer to an RGB color structure. The Movie Toolbox returns the color currently in use by the media handler. This is the blend value for blends and the transparent color for transparent operations. The Movie Toolbox supplies this value to QuickDraw when you draw in `addPin`, `subPin`, `blend`, or `transparent` mode.

ERROR CODES

Component Manager errors

SEE ALSO

You can set the graphics mode and blend color of a video media handler by calling the `SetVideoMediaGraphicsMode` function, which is described in the previous section.

Sound Media Handler Functions

Sound media handlers are responsible for interpreting and manipulating sound data. These media handlers allow you to call them directly to work with some audio settings. This section describes the functions supported by sound media handlers.

Sound media handlers maintain balance information for their audio data. You can use the `SetSoundMediaBalance` and `GetSoundMediaBalance` functions to work with a handler's balance setting.

Sample descriptions for sound media are stored in sound description structures. See "The Sound Description Structure" on page 2-79 for a discussion of the format and content of the sound description structure.

SetSoundMediaBalance

The `SetSoundMediaBalance` function sets the balance of a sound media.

```
pascal HandlerError SetSoundMediaBalance (MediaHandler mh,
                                          short balance);
```

`mH` Contains a reference to a media handler. You obtain this reference from the `GetMediaHandler` function, which is described on page 2-283.

`balance` Specifies the balance setting of the media handler as a 16-bit, fixed-point value. The high-order 8 bits contain the integer part of the value; the low-order 8 bits contain the fractional part. Valid balance values range from –1.0 to 1.0. Negative values emphasize the left sound channel, and positive values emphasize the right sound channel; a value of 0 specifies neutral balance.

ERROR CODES

Component Manager errors

GetSoundMediaBalance

The `GetSoundMediaBalance` function returns the balance of a sound media.

```
pascal HandlerError GetSoundMediaBalance (MediaHandler mh,
                                          short *balance);
```

`mH` Contains a reference to a media handler. You obtain this reference from the `GetMediaHandler` function, which is described on page 2-283.

`balance` Contains a pointer to an integer. The Movie Toolbox returns the current balance setting of the media handler as a 16-bit, fixed-point value. The high-order 8 bits contain the integer part of the value; the low-order 8 bits contain the fractional part. Valid balance values range from –1.0 to 1.0. Negative values emphasize the left sound channel, and positive values emphasize the right sound channel; a value of 0 specifies neutral balance.

ERROR CODES

Component Manager errors

Text Media Handler Functions

This section describes the functions and structure associated with the text media handler, which allows you to display text in movies. You can use text media handlers to

- add plain or styled text samples to a movie
- indicate scrolling and highlighting properties for the text
- search for text
- highlight specified text

A particular text sample has a default font, size, typeface, and color as well as a location (text box) within the track bounds to be drawn. The data format allows you to include style run information for the text. You can set flags to clip the display to the text box, inhibit automatic scaling of text as the track bounds are scaled, scroll the text, and specify if text is to be displayed at all.

The Movie Toolbox provides functions to help you add text samples to a track. You can use the `AddTextSample` function to add text to a media. The `AddTESample` function allows you to specify a TextEdit handle (which may have multiple style runs) to be added to a media. The `AddHiliteSample` function allows you to indicate highlighting for text that has just been added with the `AddTextSample` or `AddTESample` function. For more information on styled text, style runs, and TextEdit, see *Inside Macintosh: Text*.

The format of the text data that is added to the media is a 16-bit length word followed by the text. The length word specifies the number of bytes in the text. Optionally, one or more atoms of additional data may follow. An atom is structured as a 32-bit length word followed by a 32-bit type followed by some data. The length word includes the size of the data as well as the length and type fields (in other words, the size of the data plus 8).

Text atom types include the style atom (`'styl'`), the shrunken text box atom (`'tbox'`), the highlighting atom (`'hlit'`), the scroll delay atom (`'dlay'`), and the highlight color atom (`'hclr'`).

The format of the style atom is the same as TextEdit's `StScrpRec` data type. A `StScrpRec` data type is a short integer specifying the number of style runs followed by that number of `ScrpSTElement` data types, each specifying a different style run.

The shrunken text box atom is added when you set the `dfShrinkTextBoxToFit` display flag (in the `AddTextSample` or `AddTESample` function). Its format is simply the rectangle of the shrunken box (16 bytes total, including length and type).

The highlighting atom is added if the `hiliteStart` and `hiliteEnd` parameters are set appropriately in the `AddTextSample` or `AddTESample` function. When `AddHiliteSample` is called, an empty text sample (the first 2 bytes are 0) with a highlighting atom is added to the media. The format is two long integers indicating the start and end of the highlighting (16 bytes total).

The scroll delay atom specifies the scroll delay for a sample. It is a long value that specifies the delay time. It consists of 12 bytes, including the length and type fields.

The highlight color atom specifies the highlight color for a sample. Its format is an `RGBColor` data type (that is, 2 bytes red, 2 bytes green, and 2 bytes blue). It consists of 14 bytes, including the length and type fields.

The text description structure is defined as follows:

```
typedef struct TextDescription {
   long           size;             /* total size of this text
                                        description structure */
   long           type;             /* type of data in this
                                        structure such as
                                        'text' */
   long           resvd1;           /* reserved for use by
                                        Apple--set to 0 */
   long           resvd2;           /* reserved for use by
                                        Apple--set to 0 */
   short          dataRefIndex;     /* index to data references */
   long           displayFlags;     /* display flags for text */
   long           textJustification;
                                    /* text justification flags */
   RGBColor       gColor;           /* background color */
   Rect           defaultTextBox;   /* location of the text within
                                        track bounds */
   ScrpSTElement  defaultStyle;
                                    /* default style--
                                        TextEdit structure */
} TextDescription, *TextDescriptionPtr, **TextDescriptionHandle;
```

Field descriptions

`size` Defines the total size of this text description structure.

`type` Indicates the type (data type `'text'`).

`resvd1` Reserved for use by Apple. This field must be set to 0.

`resvd2` Reserved for use by Apple. This field must be set to 0.

`displayFlags` Contains the flags that specify how the text is to be displayed.

`textJustification` Contains the constant that specifies how the text is to be aligned.

`bgColor` Specifies the background color for the text display.

`defaultTextBox` Indicates the location of the text within track boundaries.

`defaultStyle` Provides a TextEdit data structure (defined by the `ScrpSTElement` data type) that specifies the default style for the text display.

The AddTextSample, AddTESample, and AddHiliteSample functions described in the sections that follow convert text into the text media format and add it to the media. To use these functions, you need to

- create a text track and media
- call the BeginMediaEdits function
- call the AddTextSample, AddTESample, or AddHiliteSample function, as appropriate
- call the EndMediaEdits function
- call the InsertMediaIntoTrack function

The movie import and export components help to get common data types (such as 'PICT' or 'snd ') into and out of movies easily. The text import component allows you to get text into a movie using the following principles:

- If you try to paste text, the text is inserted at the current position. The text import component tries to find an existing text track that fits the text.
- If no text tracks exist and there is an insertion operation, the newly created text track has the same position and size as the movie box.
- If there is an addition operation (using the Shift key), the new track is added below the movie at a height that fits the text.
- If a text track exists but the text does not fit, a new text track with sufficient height to accommodate the text is created in the same location as the existing one.
- If you hold down the Option key when you paste, the text is added in parallel at some default duration.
- If you hold down both the Option and Shift keys, the duration of the text is determined by the length of the current selection.
- If style information is on the Clipboard, it is used; otherwise, the text appears in the default 12-point application font, centered, in white on a black background.

If you want more control over how the text is added (for example, if you want to set some display flags or a new track position), your application must

1. intercept the text paste
2. instantiate its own text import component using the component type 'eat ' and component subtype 'TEXT'
3. use functions including MovieImportSetSampleDuration, MovieImportSetSampleDescription, MovieImportSetDimensions, and MovieImportSetAuxilliaryData (with 'styl' and a StScrpHandle data type)
4. call the MovieImportHandle function with the text data
5. adjust the location of the track, if desired (since the text import component may place it below the movie box)

For details on the movie import and export components, see *Inside Macintosh: QuickTime Components*.

The Movie Toolbox provides functions that allow you to search for and highlight text. You can use the `FindNextText` function to search for text in a text track, and the `HiliteTextSample` function to highlight specified text in a text track.

You can use the `SetTextProc` function (also described in this section) to specify a customized function whenever a new text sample is added to a movie. The application-defined text function `MyTextProc` is described in "Text Functions" on page 2-364.

AddTextSample

The `AddTextSample` function adds a single block of styled text to an existing media.

```
pascal ComponentResult AddTextSample (MediaHandler mh, Ptr text,
                                      unsigned long size,
                                      short fontNum,
                                      short fontSize,
                                      Style textFace,
                                      RGBColor *textColor,
                                      RGBColor *backColor,
                                      short textJustification,
                                      Rect *textBox,
                                      long displayFlags,
                                      TimeValue scrollDelay,
                                      short hiliteStart,
                                      short hiliteEnd,
                                      RGBColor *rgbHiliteColor,
                                      TimeValue duration,
                                      TimeValue *sampleTime);
```

`mh`	Specifies the media handler for the text media obtained by the `GetMediaHandler` function.
`text`	Contains a pointer to a block of text.
`size`	Indicates the size of the text block (in bytes).
`fontNum`	Indicates the number for the font in which to display the text.
`fontSize`	Indicates the size of the font.
`textFace`	Indicates the typeface or style of the text (that is, bold, italic, and so on).

`textColor` Contains a pointer to an RGB color structure specifying the color of the text.

`backColor` Contains a pointer to an RGB color structure specifying the text background color.

`textJustification`
Indicates the justification of the text. The following constants are available: `teFlushDefault`, `teCenter`, `teFlushRight`, or `teFlushLeft`. See *Inside Macintosh: Text* for details on these constants and on text alignment.

`textBox` Contains a pointer to the box within which the text is to be displayed. The box is relative to the track bounds.

`displayFlags`
Contains the text display flags.

`dfDontDisplay`
Does not display the specified sample.

`dfDontAutoScale`
Does not scale the text if the track bounds increase.

`dfClipToTextBox`
Clips to just the text box. (This is useful if the text overlays the video.)

`dfShrinkTextBoxToFit`
Recalculates size of the `textBox` parameter to just fit the given text and stores this rectangle with the text data.

`dfScrollIn`
Scrolls the text in until the last of the text is in view. This flag is associated with the `scrollDelay` parameter.

`dfScrollOut`
Scrolls text out until the last of the text is out of view. This flag is associated with the `scrollDelay` parameter. If both `dfScrollIn` and `dfScrollOut` are set, the text is scrolled in, then out.

`dfHorizScroll`
Scrolls a single line of text horizontally. If the `dfHorizScroll` flag is not set, then the scrolling is vertical.

`dfReverseScroll`
If set, scrolls vertically down, rather than up. If not set, horizontal scrolling proceeds toward the left rather than toward the right.

`scrollDelay`
Indicates the delay in scrolling associated with setting the `dfScrollIn` and `dfScrollOut` display flags. If the value of the `scrollDelay` parameter is greater than 0 and the `dfScrollIn` flag is set, the text pauses when it has scrolled all the way in for the amount of time specified

by `scrollDelay`. If the `dfScrollOut` flag is set, the pause occurs first before the text scrolls out. If both these flags are set, the pause occurs at the midpoint between scrolling in and scrolling out.

`hiliteStart`
Specifies the beginning of the text to be highlited.

`hiliteEnd` Specifies the end of the text to be highlighted. If the `hiliteEnd` parameter is greater than the `hiliteStart` parameter, then the text is highlighted from the selection specified by `hiliteStart` to `hiliteEnd`. To specify additional highlighting, you can use the `AddHiliteSample` function, described on page 2-297.

`rgbHiliteColor`
Contains a pointer to the RGB color for highlighting. If this parameter is not `nil`, then the specified color is used when highlighting the text indicated by the `hiliteStart` and `hiliteEnd` parameters. Otherwise, the default system highlighting is used.

`duration` Specifies how long the text sample should last. This duration is expressed in the media's time base.

`sampleTime`
Contains a pointer to a `TimeValue` structure. The actual media time at which the sample was added is returned here.

ERROR CODES

`invalidMedia`	–2008	This media is corrupted or invalid

File Manager errors
Memory Manager errors

AddTESample

The `AddTESample` function allows you to specify a TextEdit handle (which may contain multiple style runs) to be added to the specified media.

```
pascal ComponentResult AddTESample (MediaHandler mh, TEHandle hTE,
                                    RGBColor *backColor,
                                    short textJustification,
                                    Rect *textBox,
                                    long displayFlags,
                                    TimeValue scrollDelay,
                                    short hiliteStart,
                                    short hiliteEnd,
                                    RGBColor *rgbHiliteColor,
                                    TimeValue duration,
                                    TimeValue *sampleTime);
```

`mh` Specifies the media handler for the text media obtained by the `GetMediaHandler` function.

`hTE` A handle to a styled TextEdit structure.

`backColor` Contains a pointer to an RGB color structure specifying the text background color.

`textJustification`
Indicates the justification of the text. The following constants are available: `teFlushDefault`, `teCenter`, `teFlushRight`, or `teFlushLeft`. See *Inside Macintosh: Text* for details on these constants and on text alignment.

`textBox` Contains a pointer to the box within which the text is to be displayed. The box is relative to the track bounds.

`displayFlags`
Contains the text display flags.

`dfDontDisplay`
Does not display the specified sample.

`dfDontAutoScale`
Does not scale the text if the track bounds increase.

`dfClipToTextBox`
Clips to the text box only. (This is useful if the text overlays the video.)

`dfShrinkTextBoxToFit`
Recalculates size of the `textBox` parameter to just fit the given text and stores this rectangle with the text data.

`dfScrollIn`
Scrolls the text in until the last of the text is in view.

`dfScrollOut`
Scrolls text out until the last of the text is out of view. If both `dfScrollIn` and `dfScrollOut` are set, the text is scrolled in, then out.

`dfHorizScroll`
Scrolls a single line of text horizontally. If the `dfHorizScroll` flag is not set, then the scrolling is vertical.

`dfReverseScroll`
If set, scrolls vertically down, rather than up. If not set, horizontal scrolling proceeds toward the left rather than toward the right.

`scrollDelay`
Indicates the delay in scrolling associated with the setting of the `dfScrollIn` and `dfScrollOut` display flags. If the value of the `scrollDelay` parameter is greater than 0 and the `dfScrollIn` flag is set, the text pauses when it has scrolled all the way in for the amount of time specified by `scrollDelay`. If the `dfScrollOut` flag is set, the pause occurs first before the text scrolls out. If both these flags are set, the pause occurs at the midpoint between scrolling in and scrolling out.

hiliteStart
Specifies the beginning of the text to be highlighted.

hiliteEnd Specifies the end of the text to be highlighted. If the hiliteEnd parameter is greater than the hiliteStart parameter, then the text is highlighted from the selection specified by hiliteStart to hiliteEnd. To specify additional highlighting, you can use the AddHiliteSample function, described in the next section.

rgbHiliteColor
Contains a pointer to the RGB color for highlighting. If this parameter is not nil, then the specified color is used when highlighting the text indicated by the hiliteStart and hiliteEnd parameters. Otherwise, the default system highlight color is used.

duration Specifies how long the text sample should last. This duration is expressed in the media's time base.

sampleTime
Contains a pointer to a TimeValue structure. The actual media time at which the sample was added is returned here.

ERROR CODES

invalidMedia	–2008	This media is corrupted or invalid

File Manager errors
Memory Manager errors

AddHiliteSample

The AddHiliteSample function provides dynamic highlighting of text.

```
pascal ComponentResult AddHiliteSample (MediaHandler mh,
                                        short hiliteStart,
                                        short hiliteEnd,
                                        RGBColor *rgbHiliteColor,
                                        TimeValue duration,
                                        TimeValue *sampleTime)
```

mh Specifies the media handler for the text media obtained by the GetMediaHandler function.

hiliteStart
Indicates the beginnning of the text to be highlighted.

hiliteEnd Indicates the ending of the text to be highlighted. If the value of the hiliteStart parameter equals that of the hiliteEnd parameter, then no text is highlighted (that is, highlighting is turned off for the duration of the specified sample).

rgbHiliteColor
: Contains a pointer to the RGB color for highlighting. If this parameter is not `nil`, then the specified color is used when highlighting the text indicated by the `hiliteStart` and `hiliteEnd` parameters. Otherwise, the default system highlight color is used.

duration
: Specifies how long the text sample should last. This duration is expressed in the media's time base.

sampleTime
: Contains a pointer to a `TimeValue` structure. The actual media time at which the sample was added is returned here.

DESCRIPTION

The `AddHiliteSample` function essentially extends the duration of the text that has just been added, using the highlighting indicated by the `hiliteStart` and `hiliteEnd` parameters. You must call the `AddHiliteSample` function after calling `AddTextSample` or `AddTESample`. Since `AddHiliteSample` uses the concept of difference frames, the highlighted samples must immediately follow their associated text samples.

ERROR CODES

`invalidMedia`	–2008	This media is corrupted or invalid

File Manager errors
Memory Manager errors

FindNextText

The `FindNextText` function searches for text with a specified media handler starting at a given time.

```
pascal ComponentResult FindNextText (MediaHandler mh,
                                     Ptr text, long size,
                                     short findFlags,
                                     TimeValue startTime,
                                     TimeValue *foundTime,
                                     TimeValue *foundDuration,
                                     long *offset);
```

`mh`	Specifies the media handler for the text media obtained by the `GetMediaHandler` function.
`text`	Points to the text to be found.
`size`	Specifies the length of the text to be found.
`findFlags`	Specifies the conditions of the search. The following flags are available:
	`findTextEdgeOK` Finds sample at the given start time.
	`findTextCaseSensitive` Conducts a case-sensitive search for the text.
	`findTextReverseSearch` Searches backward for the text.
	`findTextUseOffset` Searches beginning from the value pointed to by the `offset` parameter.
	`findTextWrapAround` Conducts a wraparound search when the end or the beginning of the text is reached.
`startTime`	Indicates the time (expressed in the movie time scale) at which to begin the search.
`foundTime`	Contains a pointer to the movie time at which the text sample is found if the search is successful. Otherwise, it returns –1.
`foundDuration`	Contains a pointer to the duration of the sample (in the movie time scale) that is found if the search is successful.
`offset`	Contains a pointer to the offset of the found text from the beginning of the text portion of the sample.

DESCRIPTION

If the text sample is found, `FindNextText` returns the movie time at which it was located, the duration of the text sample, and its offset from the beginning of the text portion of the media sample.

ERROR CODES

`invalidMedia`	–2008	This media is corrupted or invalid

File Manager errors
Memory Manager errors

HiliteTextSample

When you call the `HiliteTextSample` function with a given text media handler, your application can specify selected text to be highlighted.

```
pascal ComponentResult HiliteTextSample (MediaHandler mh,
                                         TimeValue sampleTime,
                                         short hiliteStart,
                                         short hiliteEnd
                                         RGBColor *rgbHiliteColor);
```

`mh` Specifies the media handler for the text media obtained by the `GetMediaHandler` function.

`sampleTime`
Indicates a sample time (in the movie time scale) for the text to be highlighted. To turn off the highlighting in the text, pass a value of –1.

`hiliteStart`
Specifies the beginning of the text to be highlighted.

`hiliteEnd` Specifies the end of the text to be highlighted.

`rgbHiliteColor`
Contains a pointer to the RGB color for highlighting. If this parameter is not `nil`, then the specified color is used when highlighting the text indicated by the `hiliteStart` and `hiliteEnd` parameters. Otherwise, the default system highlight color is used.

DESCRIPTION

The `HiliteTextSample` function overrides any highlighting information that may already be in the specified text.

ERROR CODES

None

SEE ALSO

The `HiliteTextSample` function is useful when used in conjunction with the `FindNextText` function, described in the previous section.

SetTextProc

Your application can use the `SetTextProc` function to specify a customized function that is to be called whenever a text sample is displayed in a movie.

```
pascal ComponentResult SetTextProc (MediaHandler mh,
                                    TextMediaProcPtr TextProc,
                                    long refcon);
```

`mh` Indicates the media handler for the text media obtained by the `GetMediaHandler` function.

`TextProc` Points to the address of your customized function.

`refcon` Indicates a reference constant that will be passed to your function. Set this parameter to 0 if you don't need it.

The format of your customized text function is

```
pascal OSErr MyTextProc (Handle theText,
                         Movie theMovie,
                         short *displayFlag,
                         long refcon);
```

See "Text Functions" on page 2-364 for details on the parameters.

ERROR CODES

None

Functions for Creating File Previews

The Movie Toolbox provides two functions that allow you to create file previews. File previews contain information that gives the user an idea of a file's contents without opening the file. Typically, a file's preview is a small PICT image (called a *thumbnail*), but previews may also contain other types of information that is appropriate to the type of file being considered. For example, a text file's preview might tell the user when the file was created and what it discusses. For more information about file previews and how to display them, see "Previewing Files" on page 2-65.

Note
The `MakeFilePreview` and `AddFilePreview` functions documented in this section are not listed in the MPW `Movies.h` interface file; rather, they appear in the MPW `ImageCompression.h` interface file. u

You can use the `MakeFilePreview` function to create a preview for a file. The `AddFilePreview` function allows you to add a preview that you have created to a file.

MakeFilePreview

The `MakeFilePreview` function creates a preview for a file. You should create a preview whenever you save a movie. You specify the file by supplying a reference to its resource file. You must have opened this resource file with write permission.

```
pascal OSErr MakeFilePreview (short resRefNum,
                              ProgressProcRecordPtr progress);
```

`resRefNum` Specifies the resource file for this operation. You must have opened this resource file with write permission. If there is a preview in the specified file, the Movie Toolbox replaces that preview with a new one.

`progress` Points to a progress function. During the process of creating the preview, the Movie Toolbox may occasionally call a function you provide in order to report its progress. You can then use this information to keep the user informed.

Set this parameter to –1 to use the default progress function. If you specify a progress function, it must comply with the interface defined for Image Compression Manager progress functions (see the chapter "Image Compression Manager" in this book for more information). Set this parameter to `nil` to prevent the Movie Toolbox from calling a progress function. (For details on application-defined progress functions, see "Progress Functions," which begins on page 2-354.)

DESCRIPTION

If there is a preview in the specified file, the Movie Toolbox replaces that preview with a new one.

ERROR CODES

`paramErr`	–50	Invalid parameter specified

File Manager errors
Memory Manager errors
Resource Manager errors

AddFilePreview

The `AddFilePreview` function allows you to add a preview to a file. You must have created the preview data yourself. If the specified file already has a preview defined, the `AddFilePreview` function replaces it with the new preview.

```
pascal OSErr AddFilePreview (short resRefNum, OSType previewType,
                             Handle previewData);
```

`resRefNum` Specifies the resource file for this operation. You must have opened this resource file with write permission. If there is a preview in the specified file, the Movie Toolbox replaces that preview with a new one.

`previewType`
Specifies the resource type to be assigned to the preview. This type should correspond to the type of data stored in the preview. For example, if you have created a QuickDraw picture that you want to use as a preview for a file, you should set the `previewType` parameter to PICT.

`previewData`
Contains a handle to the preview data. For example, if the preview data is a picture, you would provide a picture handle.

DESCRIPTION

If you pass 0 for the `previewType` and `previewData` parameters, the file preview is removed.

ERROR CODES

File Manager errors
Memory Manager errors
Resource Manager errors

SEE ALSO

You can use the `MakeFilePreview` function, described in the previous section, to create a new preview for a file.

Functions for Displaying File Previews

The following section describes four functions that let you display file previews.

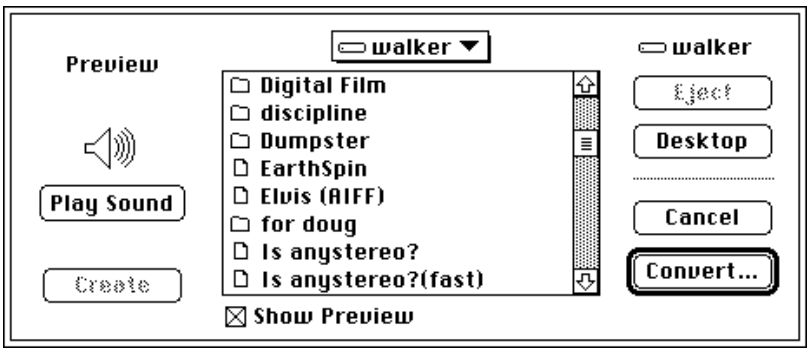
The Movie Toolbox provides two functions that allow you to display file previews in an Open dialog box in System 6 using standard file reply structures: `SFGetFilePreview` and `SFPGetFilePreview`. The Movie Toolbox also supplies two new functions that allow you to display file previews in an Open dialog box in System 7 using standard file reply structures: `StandardGetFilePreview` and `CustomGetFilePreview`.

- The `SFGetFilePreview` function corresponds to the File Manager's `SFGetFile` routine. This function is the preferred function for creating a file preview and works with either System 7 or System 6.
- The `SFPGetFilePreview` function corresponds to the File Manager's `SFPGetFile` routine.
- The `StandardGetFilePreview` function corresponds to the File Manager's `StandardGetFile` routine.
- The `CustomGetFilePreview` function corresponds to the File Manager's `CustomGetFile` routine. This function is available only in System 7.

All of these functions take the same parameters as their existing counterparts with the addition of a `where` parameter that allows you to specify the location of the upper-left corner of the dialog box. See *Inside Macintosh: Files* for information on the `SFGetFile`, `SFPGetFile`, `StandardGetFile`, and `CustomGetFile` routines.

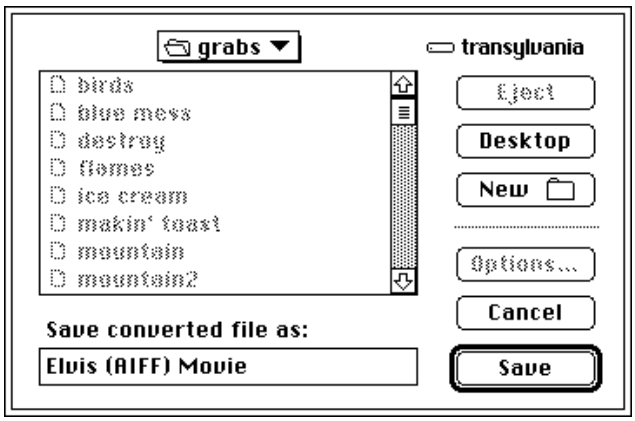
The `SFGetFilePreview`, `SFPGetFilePreview`, `StandardGetFilePreview`, and `CustomGetFilePreview` functions allow the user to automatically convert files to movies if your application requests movies. If there is a file that can be converted into a movie file using a movie import component, then the file is shown in the Standard File dialog box in addition to any movies. When the user selects the file, the Open button changes to a Convert button. Figure 2-41 provides an example of this dialog box.

Figure 2-41 Dialog box showing automatic file-to-movie conversion option



Choosing Convert displays a dialog box that allows the user to choose where the converted file should be saved. Figure 2-42 shows this dialog box.

Figure 2-42 Dialog box for saving a movie converted from a file



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

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

Note
The functions described in this section do not appear in the MPW interface file `Movies.h`; rather, they are listed in `ImageCompression.h`. u

SFGetFilePreview

The SFGetFilePreview function allows you to display file previews in an Open dialog box using a standard file reply structure. This is the preferred function for displaying a file preview and it works with either System 7 or System 6.

```
pascal void SFGetFilePreview (Point where,
                              ConstStr255Param prompt,
                              FileFilterProcPtr fileFilter,
                              short numTypes, SFTypeList typeList,
                              DlgHookProcPtr dlgHook,
                              SFReply *reply);
```

`where` Specifies the location of the upper-left corner of the dialog box in global coordinates. If you set this point to (–1, –1), the Movie Toolbox centers the dialog box on the main screen. If you set this point to (–2, –2), the Movie Toolbox centers the dialog box on the screen that has the best display characteristics.

`prompt` This parameter is ignored; it is included for historical reasons only.

`fileFilter`

Points to a function that filters the files that are displayed to the user in the dialog box. This is an optional function provided by your application; if you do not want to supply a filter function, set this parameter to `nil`. The `SFGetFilePreview` function uses this parameter along with the `numTypes` and `typeList` parameters to determine which files appear in the dialog box.

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 (see *Inside Macintosh: Files* for more information). Your function returns a Boolean value indicating whether to display the file. Set the Boolean value to `false` to cause the file to be displayed.

Your function must provide the following interface:

```
pascal Boolean MyFileFilter (ParmBlkPtr parmBlock);
```

See "File Filter Functions" on page 2-360 for details.

`numTypes` Specifies the number of file types in the array specified by the `typeList` parameter (a number between 1 and 4). Set this parameter to –1 to display all files.

typeList Specifies an array of file types to be displayed to the user. The SFGetFilePreview function only displays files whose type matches an entry in this array (unless you set the numTypes parameter to –1; in this case, the function displays all files to the user). The SFTypeList data type is defined as follows:

```
typedef OSType SFTypeList[4];
```

dlgHook Specifies a pointer to a custom dialog function. You can use this parameter to support a custom dialog box function you have supplied. If you are not supplying a custom dialog box function, set this parameter to nil. Your custom dialog function must present the following interface:

```
pascal short MyDlgHook (short item,
                        DialogPtr theDialog,
                        Ptr myDataPtr);
```

For more information about using custom dialog box functions with the SFGetFilePreview function, see "Custom Dialog Functions" on page 2-360.

reply Contains a pointer to a standard file reply structure that is to receive information about the user's selection. See *Inside Macintosh: Files* for more information about reply structures.

DESCRIPTION

The SFGetFilePreview function presents an Open dialog box to the user and allows the user to view file previews during the dialog. This function corresponds to the File Manager's SFGetFile routine. See *Inside Macintosh: Files* for a complete description of the SFGetFile routine.

The SFGetFilePreview function takes the same parameters as its existing counterpart with the addition of a where parameter that allows you to specify the location of the dialog box.

The SFGetFilePreview function automatically converts files to movies if your application requests movies. If a file could be converted into a movie file using a movie import component, then the file is shown in the Standard File dialog box. See Figure 2-41 on page 2-304 for the dialog box with an automatic file-to-movie conversion option and Figure 2-42 on page 2-305 for the dialog box for saving a movie converted from a file.

Note

The SFGetFilePreview function does not appear in the MPW interface file Movies.h; rather, it's listed in ImageCompression.h. u

SFPGetFilePreview

The SFPGetFilePreview function allows you to display file previews in an Open dialog box using a standard file reply structure. This function differs from SFGetFilePreview in that you can provide a custom dialog box with any resource type and you can specify a modal-dialog filter function that allows you to gain greater control over the user interface.

```
pascal void SFPGetFilePreview (Point where,
                               ConstStr255Param prompt,
                               FileFilterProcPtr fileFilter,
                               short numTypes,
                               SFTypeList typeList,
                               DlgHookProcPtr dlgHook,
                               SFReply *reply, short dlgID,
                               ModalFilterProcPtr filterProc);
```

where Specifies the location of the upper-left corner of the dialog box in global coordinates. If you set this point to (–1, –1), the Movie Toolbox centers the dialog box on the main screen. If you set this point to (–2, –2), the Movie Toolbox centers the dialog box on the screen that has the best display characteristics.

prompt This parameter is ignored; it is included for historical reasons only.

fileFilter

Points to a function that filters the files that are displayed to the user in the dialog box. This is an optional function provided by your application; if you do not want to supply a filter function, set this parameter to nil. The SFGetFilePreview function uses this parameter along with the numTypes and typeList parameters to determine which files appear in the dialog box.

If this parameter is not nil, SFPGetFilePreview calls the function for each file to determine whether to display the file to the user. The SFPGetFilePreview function supplies you with the information it receives from the File Manager's GetFileInfo routine (see *Inside Macintosh: Files* for more information). Your function returns a Boolean value indicating whether to display the file. Set the Boolean value to false to cause the file to be displayed. See "File Filter Functions," which begins on page 2-360, for details on file filter functions.

Your function must provide the following interface:

```
pascal Boolean MyFileFilter (ParmBlkPtr parmBlock);
```

numTypes Specifies the number of file types in the array specified by the typeList parameter. Specify a number between 1 and 4. Set this parameter to –1 to display all files.

typeList Specifies an array of file types to be displayed to the user. The SFGetFilePreview function only displays files whose type matches an entry in this array (unless you set the numTypes parameter to –1; in this case, the function displays all files to the user). The SFTypeList data type is defined as follows:

```
typedef OSType SFTypeList[4];
```

dlgHook Points to a custom dialog box function. 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 (the dialog template's resource type must be set to 'DLOG'; you must also supply an item list in a 'DITL' resource). You specify the dialog template's resource ID with the dlgID parameter. If you are not supplying a custom dialog function in this manner, set this parameter to nil.

Your custom dialog box function must present the following interface:

```
pascal short MyDlgHook (short item,
                        DialogPtr theDialog,
                        Ptr myDataPtr);
```

See "Custom Dialog Functions" on page 2-360 for more information on using custom dialog functions with the SFPGetFilePreview function.

reply Contains a pointer to a standard file reply structure that is to receive information about the user's selection. See *Inside Macintosh: Files* for more information about reply structures.

dlgID Specifies the resource ID of your custom dialog template. You can use this parameter to specify a custom dialog template resource that has a resource type that differs from the standard value. Set this parameter to 0 to use the standard template.

filterProc

Points to your modal-dialog filter function. This function gives you greater control over the interface presented to the user. Your modal-dialog filter function must present the following interface:

```
pascal Boolean MyModalFilter (DialogPtr theDialog,
                              EventRecord* theEvent,
                              short itemHit,
                              Ptr myDataPtr);
```

See "Modal-Dialog Filter Functions" beginning on page 2-362 for details.

DESCRIPTION

The SFPGetFilePreview function presents an Open dialog box to the user and allows the user to view file previews during the dialog. This function corresponds to the File Manager's SFPGetFile routine. The SFPGetFilePreview function takes the same parameters as its existing counterpart with the addition of a where parameter that allows you to specify the location of the dialog box. See *Inside Macintosh: Files* for a complete description of the SFPGetFile routine and for more information about the parameters to this function.

The SFPGetFilePreview function automatically converts files to movies if your application requests movies. If a file could be converted into a movie file using a movie import component, then the file is shown in the Standard File dialog box. See Figure 2-41 on page 2-304 for the dialog box with an automatic file-to-movie conversion option and Figure 2-42 on page 2-305 for the dialog box for saving a movie converted from a file.

Note
The SFPGetFilePreview function does not appear in the MPW interface file Movies.h; rather, it's listed in ImageCompression.h. u

StandardGetFilePreview

The SFPGetFilePreview function allows you to display file previews in an Open dialog box using a standard file reply structure.

```
pascal void StandardGetFilePreview (FileFilterProcPtr fileFilter,
                                    short numTypes,
                                    SFTypeList typeList,
                                    StandardFileReply *reply);
```

fileFilter

Points to a function that filters the files that are displayed to the user in the dialog box. This is an optional function provided by your application; if you do not want to supply a filter function, set this parameter to nil. The StandardGetFilePreview function uses this parameter along with the numTypes and typeList parameters to determine which files appear in the dialog box.

If this parameter is not nil, StandardGetFilePreview calls the function for each file to determine whether to display the file to the user. The StandardGetFilePreview function supplies you with information identifying the file (see *Inside Macintosh: Files* for

more information about the format of this parameter data). Your function returns a Boolean value indicating whether to display the file. Set the Boolean value to `false` to cause the file to be displayed.

Your function must provide the following interface:

```
pascal Boolean MyFileFilter (ParmBlkPtr parmBlock);
```

`numTypes` Specifies the number of file types in the array specified by the `typeList` parameter (a number between 1 and 4). Set this parameter to –1 to display all files.

`typeList` Specifies an array of file types to be displayed to the user. The `StandardGetFilePreview` function only displays files whose type matches an entry in this array (unless you set the `numTypes` parameter to –1; in this case, the function displays all files to the user). The `SFTypeList` data type is defined as follows:

```
typedef OSType SFTypeList[4];
```

`reply` Contains a pointer to a reply structure that is to receive information about the user's selection. See *Inside Macintosh: Files* for more information about reply structures.

DESCRIPTION

The `StandardGetFilePreview` function presents an Open dialog box to the user and allows the user to view file previews. This function corresponds to the File Manager's `StandardGetFile` routine. See *Inside Macintosh: Files* for a comprehensive description of that routine and for more information about the parameters to this function. The `StandardGetFilePreview` function takes the same parameters as its existing counterpart with the addition of a `where` parameter that allows you to specify the location of the dialog box.

The `StandardGetFilePreview` function automatically converts files to movies if your application requests movies. If a file could be converted into a movie file using a movie import component, then the file is shown in the Standard File dialog box. See Figure 2-41 on page 2-304 for the dialog box with an automatic file-to-movie conversion option and Figure 2-42 on page 2-305 for the dialog box for saving a movie converted from a file.

Note

The `StandardGetFilePreview` function does not appear in the MPW interface file `Movies.h`; rather, it's listed in `ImageCompression.h`. u

CustomGetFilePreview

The CustomGetFilePreview function 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.

Note
The CustomGetFilePreview function is available only in System 7. u

```
pascal void CustomGetFilePreview (FileFilterYDProcPtr fileFilter,
                                  short numTypes, SFTypeList
                                  typeList, StandardFileReply
                                  *reply, short dlgID,
                                 Point where,
                                 DlgHookYDProcPtr dlgHook,
                                 ModalFilterYDProcPtr filterProc,
                                 short *activeList,
                                 ActivateYDProcPtr activateProc,
                                 void *yourDataPtr);
```

fileFilter
Points to a function that filters the files that are displayed to the user in the dialog box. This is an optional function provided by your application; if you do not want to supply a filter function, set this parameter to nil. The CustomGetFilePreview function uses this parameter along with the numTypes and typeList parameters to determine which files appear in the dialog box.

If this parameter is not nil, CustomGetFilePreview calls the function for each file to determine whether to display the file to the user. The CustomGetFilePreview function supplies you with information identifying the file (see *Inside Macintosh: Files* for more information about the format of this parameter data). Your function returns a Boolean value indicating whether to display the file. Set the Boolean value to false to cause the file to be displayed.

Your function must provide the following interface:

```
pascal Boolean MyFileFilter (ParmBlkPtr parmBlock);
```

numTypes Specifies the number of file types in the array specified by the typeList parameter (a number between 1 and 4). Set this parameter to –1 to display all files.

typeList Specifies an array of file types to be displayed to the user. The CustomGetFilePreview function only displays files whose type matches an entry in this array (unless you set the numTypes parameter

to –1; in this case, the function displays all files to the user). The `SFTypeList` data type is defined as follows:

```
typedef OSType SFTypeList[4];
```

`reply` Contains a pointer to a reply structure that is to receive information about the user's selection. See *Inside Macintosh: Files* for more information about reply structures.

`dlgID` Specifies the resource ID of your custom dialog template. You can use this parameter to specify a custom dialog template resource that has a resource type that differs from the standard value. Set this parameter to 0 to use the standard template.

`where` Specifies the location of the upper-left corner of the dialog box in global coordinates. If you set this point to (–1, –1), the Movie Toolbox centers the dialog box on the main screen. If you set this point to (–2, –2), the Movie Toolbox centers the dialog box on the screen that has the best display characteristics.

`dlgHook` Points to a custom dialog function. 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`. For more information about using custom dialog functions with the `CustomGetFile` routine, see *Inside Macintosh: Files*. For details on the parameters of the custom dialog box function, see "Custom Dialog Functions" on page 2-360.

Your dialog hook function must present the following interface:

```
pascal short MyDlgHook (short item, DialogPtr
                        theDialog, Ptr myDataPtr);
```

`filterProc`

Points to your modal-dialog filter function. This function gives you greater control over the interface presented to the user. See *Inside Macintosh: Files* for more information about using modal-dialog filter functions with `CustomGetFile`.

Your modal-dialog filter function must present the following interface.

```
pascal Boolean MyModalFilter (DialogPtr theDialog,
                              EventRecord* theEvent,
                              short itemHit,
                              Ptr myDataPtr);
```

For details on the application-defined modal-dialog filter, see "Modal-Dialog Filter Functions" beginning on page 2-362.

activeList

Contains a pointer to a list of all items in the dialog box that can be activated—that is, made the target of keyboard input. The list is stored as an array of integers. The first integer must contain the number of items in the array (not including this count value). The remaining array entries must contain item numbers that specify valid targets of keyboard input, in the order in which the items are to be activated. Set this parameter to `nil` to direct all keyboard input to the displayed list of filenames.

activateProc

Points to your activation function, which controls the highlighting of any items whose shape is known only by your application. See *Inside Macintosh: Files* for more information about standard file activation functions.

Your function must present the following interface:

```
pascal void MyActivateProc (DialogPtr theDialog,
                            short itemNo,
                            Boolean activating,
                            Ptr myDataPtr);
```

yourDataPtr

Contains a pointer to optional data that is supplied by your application to your callback functions. When the `CustomGetFilePreview` function calls any of your callback functions, it places this data on the stack, making it available to your functions. Set this parameter to `nil` if you are not supplying any optional data.

DESCRIPTION

The `CustomGetFilePreview` function is available only if the value of the Gestalt selector `gestaltStandardFileAttr` is `true`. (See *Inside Macintosh: Overview* for more information about this selector.) This function corresponds to the File Manager's `CustomGetFile` routine. The `CustomGetFilePreview` function takes the same parameters as its existing counterpart with the addition of a `where` parameter that allows you to specify the location of the dialog box. See *Inside Macintosh: Files* for a complete description of the `CustomGetFile` routine and for more information about the parameters to this function.

The `CustomGetFilePreview` function automatically converts files to movies if your application requests movies. If a file could be converted into a movie file using a movie import component, then the file is shown in the Standard File dialog box. See Figure 2-41 on page 2-304 for the dialog box with an automatic file-to-movie conversion option and Figure 2-42 on page 2-305 for the dialog box for saving a movie converted from a file.

Note

The `CustomGetFilePreview` function does not appear in the MPW interface file `Movies.h`; rather, it's listed in `ImageCompression.h`. u

Time Base Functions

The Movie Toolbox provides a number of functions that allow you to work with time bases. A QuickTime time base defines the time coordinate system of a movie. However, you can also use QuickTime time bases to provide general timing services. This section describes the functions that allow your application to work with time bases. For a complete description of QuickTime time bases, see "Introduction to Movies" beginning on page 2-5.

This section has been divided into the following topics:

- "Creating and Disposing of Time Bases" describes how to create and dispose of time bases and how to assign a time base to a movie
- "Working With Time Base Values" discusses functions that allow your application to work with the contents of a time base
- "Working With Times" describes a number of functions that allow you to convert times between time bases and to perform simple arithmetic on time values
- "Time Base Callback Functions" describes the functions your application may use to condition a time base to invoke functions your application provides

Note
Time base functions do not change the value of the Movie Toolbox sticky error value. u

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. These functions are noted in the sections that follow. u

NewTimeBase

The `NewTimeBase` function allows your application to obtain a new time base. This function returns a reference to the new time base. Your application must use that reference with other time base functions.

```
pascal TimeBase NewTimeBase (void);
```

DESCRIPTION

The `NewTimeBase` function returns a reference to the new time base.

This function sets the rate of the time base to 0, the start time to its minimum value, the time value to 0, and the stop time to its maximum value.

This function assigns the default clock component to the new time base. If you want to assign a different clock component or a master time base to the new time base, use the `SetTimeBaseMasterClock` or `SetTimeBaseMasterTimeBase` functions, which are described on page 2-318 and page 2-320, respectively.

SPECIAL CONSIDERATIONS

The `NewTimeBase` function uses the Memory Manager, so your application must not call it at interrupt time.

ERROR CODES

None

DisposeTimeBase

The `DisposeTimeBase` function allows your application to dispose of a time base once you are finished with it.

```
pascal void DisposeTimeBase (TimeBase tb);
```

`tb` Specifies the time base for this operation. Your application obtains this time base identifier from the `NewTimeBase` function described in the previous section.

DESCRIPTION

The `DisposeTimeBase` function cancels and disposes of any pending callback events that are associated with the time base.

SPECIAL CONSIDERATIONS

Note that the `DisposeTimeBase` function uses the Memory Manager; therefore, you should not call this function at interrupt time.

ERROR CODES

None

SetMovieMasterClock

You can use the `SetMovieMasterClock` function to assign a clock component to a movie. Do not use the `SetTimeBaseMasterClock` function to assign a clock component to a movie.

```
pascal void SetMovieMasterClock (Movie theMovie,
                                 Component clockMeister,
                                 const TimeRecord *slaveZero);
```

`theMovie` Specifies the movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

`clockMeister` Specifies the clock component to be assigned to this movie. Your application can obtain this component identifier from the Component Manager's `FindNextComponent` routine (see the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for more information about this routine).

`slaveZero` Contains a pointer to the time, in the clock's time scale, that corresponds to a 0 time value for the movie. This parameter allows you to set an offset between the clock component and the time base of the movie. Set this parameter to `nil` if there is no offset.

ERROR CODES

None

SetMovieMasterTimeBase

You can use the `SetMovieMasterTimeBase` function to assign a master time base to a movie. Do not use the `SetTimeBaseMasterTimeBase` function (described on page 2-320) to assign a time base to a movie.

```
pascal void SetMovieMasterTimeBase (Movie theMovie, TimeBase tb,
                                    const TimeRecord *slaveZero);
```

`theMovie` Specifies the movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

`tb` Specifies the master time base to be assigned to this movie. Your application obtains this time base identifier from the `NewTimeBase` function (described on page 2-316).

`slaveZero` Contains a pointer to the time, in the time scale of the master time base, that corresponds to a 0 time value for the movie. This parameter allows you to set an offset between the movie and the master time base. Set this parameter to `nil` if there is no offset.

SPECIAL CONSIDERATIONS

The `SetMovieMasterTimeBase` function cannot be called at interrupt time.

ERROR CODES

None

SetTimeBaseMasterClock

You can use the `SetTimeBaseMasterClock` function to assign a clock component to a time base. A time base derives its time from either a clock component or from another time base. Do not use this function to assign a clock to a movie's time base.

```
pascal void SetTimeBaseMasterClock (TimeBase slave,
                                    Component clockMeister,
                                    const TimeRecord *slaveZero);
```

`slave` Specifies the time base for this operation. Your application obtains this time base identifier from the `NewTimeBase` function (described on page 2-316).

`clockMeister`
Specifies the clock component to be assigned to this time base. Your application can obtain this component identifier from the Component Manager's `FindNextComponent` routine (see the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for more information about this routine).

`slaveZero` Contains a pointer to the time, in the clock's time scale, that corresponds to a 0 time value for the slave time base. This parameter allows you to set an offset between the time base and the clock component. Set this parameter to `nil` if there is no offset.

SPECIAL CONSIDERATIONS

The `SetTimeBaseMasterClock` function cannot be called at interrupt time.

ERROR CODES

`invalidMovie`	–2010	This movie is corrupted or invalid

SEE ALSO

You can use the `GetTimeBaseMasterClock` function, which is described in the next section, to determine the clock component that is assigned to a time base.

GetTimeBaseMasterClock

You can use the `GetTimeBaseMasterClock` function to determine the clock component that is assigned to a time base. A time base derives its time from either a clock component or from another time base. If a time base derives its time from a clock component, you can use this function to obtain the component instance of the clock component.

```
pascal ComponentInstance GetTimeBaseMasterClock (TimeBase tb);
```

`tb` Specifies the time base for this operation. Your application obtains this time base identifier from the `NewTimeBase` function (described on page 2-316).

DESCRIPTION

The `GetTimeBaseMasterClock` function returns a reference to a component instance of the clock component that provides a time source to this time base.

Note
The Component Manager allows a single component to serve multiple client applications at the same time. Each client application has a unique access path to the component. These access paths are called **connections.** You identify a component connection by specifying a **component instance.** The Component Manager provides this component instance to your application when you open a connection to a component. The component maintains separate status information for each open connection. u

Do not close this connection—the time base is using the connection to maintain its time source. If a clock component is not assigned to the time base, this function sets the returned reference to `nil`. In this case, the time base relies on another time base for its time source. Use the `GetTimeBaseMasterTimeBase` function, which is described on page 2-321, to obtain the time base reference to that master time base.

ERROR CODES

None

SEE ALSO

You can use the `SetTimeBaseMasterClock` function, which is described on page 2-318, to assign a clock component to a time base.

SetTimeBaseMasterTimeBase

You can use the `SetTimeBaseMasterTimeBase` function to assign a master time base to a time base. A time base derives its time from either a clock component or another time base. Do not use this function to assign a master time base to a movie's time base.

```
pascal void SetTimeBaseMasterTimeBase (TimeBase slave,
                                       TimeBase master,
                                    const TimeRecord *slaveZero);
```

`slave` Specifies the time base for this operation. Your application obtains this time base identifier from the `NewTimeBase` function (described on page 2-316).

`master` Specifies the master time base to be assigned to this time base. Your application obtains this time base identifier from the `NewTimeBase` function.

`slaveZero` Contains a pointer to the time, in the time scale of the master time base, that corresponds to a 0 time value for the slave time scale. This parameter allows you to set an offset between the time base and the master time base. Set this parameter to `nil` if there is no offset.

ERROR CODES

None

SEE ALSO

You can use the `GetTimeBaseMasterTimeBase` function, which is described in the next section, to determine the master time base that is assigned to a time base.

GetTimeBaseMasterTimeBase

You can use the `GetTimeBaseMasterTimeBase` function to determine the master time base that is assigned to a time base. A time base derives its time from either a clock component or from another time base. If a time base derives its time from another time base, you can use this function to obtain the identifier for that master time base.

```
pascal TimeBase GetTimeBaseMasterTimeBase (TimeBase tb);
```

`tb` Specifies the time base for this operation. Your application obtains this time base identifier from the `NewTimeBase` function (described on page 2-316).

DESCRIPTION

The `GetTimeBaseMasterTimeBase` function returns a reference to the master time base that provides a time source to this time base. If a master time base is not assigned to the time base, this function sets the returned reference to `nil`. In this case, the time base relies on a clock component for its time source. Use the `GetTimeBaseMasterClock` function, which is described on page 2-319, to obtain the component instance reference to that clock component.

ERROR CODES

None

SEE ALSO

You can use the `SetTimeBaseMasterTimeBase` function, which is described in the previous section, to assign a master time base to a time base.

SetTimeBaseZero

You can use the `SetTimeBaseZero` function to change the offset from a time base to either its master time base or its clock component. You establish the initial offset when you assign the time base to its time source.

```
pascal void SetTimeBaseZero (TimeBase tb, TimeRecord *zero);
```

`tb` Specifies the time base for this operation. Your application obtains this time base identifier from the `NewTimeBase` function (described on page 2-316).

`zero` Contains a pointer to the time that corresponds to a 0 time value for the slave time scale. This parameter allows you to set an offset between the time base and its time source. Set this parameter to `nil` if there is no offset.

ERROR CODES

None

SEE ALSO

You can use the `SetTimeBaseMasterClock` function (described on page 2-318) to assign a time base to a clock component.

You can use the `SetTimeBaseMasterTimeBase` function (described on page 2-320) to assign a time base to a master time base.

Working With Time Base Values

Every time base contains a rate, a start time, a stop time, a current time, and some status information. The Movie Toolbox provides a number of functions that allow your application to work with the contents of a time base. This section describes those functions.

The `GetTimeBaseTime` function lets you retrieve the current time value of a time base. You can set the current time value by calling the `SetTimeBaseTime` function—this function requires you to provide a time structure. Alternatively, you can set the current time based on a time value by calling the `SetTimeBaseValue` function.

You can determine the rate of a time base by calling the `GetTimeBaseRate` function. You can set the rate of a time base by calling the `SetTimeBaseRate` function. You can determine the effective rate of a specified time base (relative to the master time base to which it is subordinate) by calling the `GetTimeBaseEffectiveRate` function.

You can retrieve the start time of a time base by calling the `GetTimeBaseStartTime` function. You can set the start time of a time base by calling the `SetTimeBaseStartTime` function. Similarly, you can use the `GetTimeBaseStopTime` and `SetTimeBaseStopTime` functions to work with the stop time of a time base.

The Movie Toolbox provides functions that allow you to work with the status information of a time base. The `GetTimeBaseStatus` function allows you to read the current status of a time base. The `GetTimeBaseFlags` function helps you obtain the control flags of a time base. You can set these flags by calling the `SetTimeBaseFlags` function.

SetTimeBaseTime

The `SetTimeBaseTime` function allows your application to set the current time of a time base. You must specify the new time in a time structure.

```
pascal void SetTimeBaseTime (TimeBase tb, const TimeRecord *tr);
```

`tb` Specifies the time base for this operation. Your application obtains this time base identifier from the `NewTimeBase` function (described on page 2-316).

`tr` Contains a pointer to a time structure that contains the current time value.

DESCRIPTION

If you set the current time of a time base that is the master time base for other time bases, the current times in all the dependent time bases are changed appropriately. If you change the current time in a time base that relies on a master time base, the Movie Toolbox changes the offset between the time base and the master time base—the master time base is not affected.

ERROR CODES

None

SEE ALSO

You can set the current time of a time base from a time value by calling the `SetTimeBaseValue` function, which is described in the next section.

SetTimeBaseValue

The `SetTimeBaseValue` function allows your application to set the current time of a time base. You must specify the new time as a time value.

```
pascal void SetTimeBaseValue (TimeBase tb, TimeValue t,
                              TimeScale s);
```

`tb` Specifies the time base for this operation. Your application obtains this time base identifier from the `NewTimeBase` function (described on page 2-316).

`t` Specifies the new time value.

`s` Specifies the time scale of the new time value.

DESCRIPTION

If you set the current time of a time base that is the master time base for other time bases, the current times in all the dependent time bases are changed appropriately. If you change the current time in a time base that relies on a master time base, the Movie Toolbox changes the offset between the time base and the master time base—the master time base is not affected.

ERROR CODES

None

SEE ALSO

You can set the current time of a time base from a time structure by calling the `SetTimeBaseTime` function, which is described in the previous section.

GetTimeBaseTime

Your application can use the `GetTimeBaseTime` function to obtain the current time value from a time base. You can specify the time scale in which to return the time value.

```
pascal TimeValue GetTimeBaseTime (TimeBase tb, TimeScale s,
                                  TimeRecord *tr);
```

`tb` Specifies the time base for this operation. Your application obtains this time base identifier from the `NewTimeBase` function (described on page 2-316).

s	Specifies the time scale in which to return the current time value. Set this parameter to 0 to retrieve the time in the preferred time scale of the time base.
tr	Contains a pointer to a time structure that is to receive the current time value. This is an optional parameter. If you do not want the time value represented in a time structure, set this parameter to nil.

DESCRIPTION

The GetTimeBaseTime function returns a time value that contains the current time from the specified time base in the specified time scale. The function returns this value even if you specify a time structure with the tr parameter.

ERROR CODES

None

SEE ALSO

You can set the current time of a time base by calling either the SetTimeBaseTime or SetTimeBaseValue functions, which are described on page 2-323 and page 2-324, respectively.

SetTimeBaseRate

The SetTimeBaseRate function allows your application to set the rate of a time base.

```
pascal void SetTimeBaseRate (TimeBase tb, Fixed r);
```

tb	Specifies the time base for this operation. Your application obtains this time base identifier from the NewTimeBase function (described on page 2-316).
r	Specifies the rate of the time base.

DESCRIPTION

You can determine the number of time units that pass each second for a time base by multiplying its rate by the time scale of its time coordinate system. For example, if you set the rate of a time base to 2 and the time base has a time scale of 2, that time base passes through 4 units of its time each second.

Rates may be set to negative values. Negative rates cause time to move backward for the time base.

ERROR CODES

None

SEE ALSO

You can retrieve the rate of a time base by calling the `GetTimeBaseRate` function, which is described in the next section.

GetTimeBaseRate

The `GetTimeBaseRate` function allows your application to retrieve the rate of a time base.

Rates may be set to negative values. Negative rates cause time to move backward for the time base.

```
pascal Fixed GetTimeBaseRate (TimeBase tb);
```

`tb` Specifies the time base for this operation. Your application obtains this time base identifier from the `NewTimeBase` function (described on page 2-316).

DESCRIPTION

The `GetTimeBaseRate` function returns the current rate of the time base as a fixed-point number. This rate value may be nonzero even if the time base has stopped, because it has reached its stop time.

ERROR CODES

None

GetTimeBaseEffectiveRate

The `GetTimeBaseEffectiveRate` function returns the effective rate at which the specified time base is moving, relative to its master clock.

```
pascal Fixed GetTimeBaseEffectiveRate (TimeBase tb);
```

tb Specifies the time base for this operation. Your application obtains this time base identifier from the NewTimeBase function (described on page 2-316).

DESCRIPTION

The GetTimeBaseEffectiveRate function is useful when you need to make scheduling decisions based on the rate of a time base—for example, when you are writing a media handler. (For more on media handlers, see *Inside Macintosh: QuickTime Components*.) By calling GetTimeBaseEffectiveRate rather than the GetTimeBaseRate function (described in the previous section), you can easily take into account any time base subordination that may be in effect.

SetTimeBaseStartTime

You can set the start time of a time base by calling the SetTimeBaseStartTime function. The start time defines the time base's minimum time value. You must specify the new start time in a time structure.

```
pascal void SetTimeBaseStartTime (TimeBase tb,
                                  const TimeRecord *tr);
```

tb Specifies the time base for this operation. Your application obtains this time base identifier from the NewTimeBase function (described on page 2-316).

tr Contains a pointer to a time structure that contains the start time value.

DESCRIPTION

Do not use this function to restrict the Movie Toolbox to a portion of a movie—use the SetMovieActiveSegment function (described on page 2-136) instead.

ERROR CODES

None

SEE ALSO

You can determine the start time of a time base by calling the GetTimeBaseStartTime function, which is described in the next section.

GetTimeBaseStartTime

You can determine the start time of a time base by calling the `GetTimeBaseStartTime` function.

```
pascal TimeValue GetTimeBaseStartTime (TimeBase tb, TimeScale s,
                                       TimeRecord *tr);
```

`tb` Specifies the time base for this operation. Your application obtains this time base identifier from the `NewTimeBase` function (described on page 2-316).

`s` Specifies the time scale in which to return the start time.

`tr` Contains a pointer to a time structure that is to receive the start time. This is an optional parameter. If you do not want the time value represented in a time structure, set this parameter to `nil`.

DESCRIPTION

The `GetTimeBaseStartTime` returns a time value that contains the start time from the specified time base in the specified time scale. The function returns this value even if you specify a time structure with the `tr` parameter.

ERROR CODES

None

SEE ALSO

You can set the start time of a time base by calling the `SetTimeBaseStartTime` function, which is described in the previous section.

SetTimeBaseStopTime

You can set the stop time of a time base by calling the `SetTimeBaseStopTime` function. The stop time defines the time base's maximum time value. You must specify the new stop time in a time structure.

```
pascal void SetTimeBaseStopTime (TimeBase tb,
                                 const TimeRecord *tr);
```

`tb` Specifies the time base for this operation. Your application obtains this time base identifier from the `NewTimeBase` function (described on page 2-316).

`tr` Contains a pointer to a time structure that contains the stop time value.

DESCRIPTION

Do not use the `SetTimeBaseStopTime` function to restrict the Movie Toolbox to a portion of a movie—use the `SetMovieActiveSegment` function (described on page 2-136) instead.

ERROR CODES

None

SEE ALSO

You can determine the stop time of a time base by calling the `GetTimeBaseStopTime` function, which is described in the next section.

GetTimeBaseStopTime

You can determine the stop time of a time base by calling the `GetTimeBaseStopTime` function.

```
pascal TimeValue GetTimeBaseStopTime (TimeBase tb, TimeScale s,
                                      TimeRecord *tr);
```

`tb` Specifies the time base for this operation. Your application obtains this time base identifier from the `NewTimeBase` function (described on page 2-316).

`s` Specifies the time scale in which to return the stop time.

`tr` Contains a pointer to a time structure that is to receive the stop time. This is an optional parameter. If you do not want the time value represented in a time structure, set this parameter to `nil`.

DESCRIPTION

The `GetTimeBaseStopTime` returns a time value that contains the stop time from the specified time base in the specified time scale. The function returns this value even if you specify a time structure with the `out` parameter.

ERROR CODES

None

SEE ALSO

You can set the stop time of a time base by calling the `SetTimeBaseStopTime` function, which is described in the previous section.

SetTimeBaseFlags

The `SetTimeBaseFlags` function allows your application to set the contents of the control flags of a time base.

```
pascal void SetTimeBaseFlags (TimeBase tb, long timeBaseFlags);
```

`tb` Specifies the time base for this operation. Your application obtains this time base identifier from the `NewTimeBase` function (described on page 2-316).

`timeBaseFlags`
Specifies the control flags for this time base. The following flags are defined. You may set only one flag to 1 (be sure to set unused flags to 0):

`loopTimeBase`
Indicates whether the time base loops. If you set this flag to 1 and the rate is positive, the time base loops back and restarts from its start time when it reaches its stop time. If you set this flag to 1 and the rate is negative, the time base loops to its stop time. If you set the flag to 0, the movie stops when it reaches the end.

`palindromeLoopTimeBase`
Indicates whether the time base loops in a palindrome fashion. **Palindrome looping** causes a time base to move alternately forward and backward. Set this flag to 1 to cause the time base to loop in this manner.

ERROR CODES

None

SEE ALSO

You can retrieve the control flags of a time base by calling the `GetTimeBaseFlags` function, which is described in the next section.

GetTimeBaseFlags

The `GetTimeBaseFlags` function allows your application to obtain the contents of the control flags of a time base.

```
pascal long GetTimeBaseFlags (TimeBase tb);
```

`tb` Specifies the time base for this operation. Your application obtains this time base identifier from the `NewTimeBase` function (described on page 2-316).

DESCRIPTION

The `GetTimeBaseFlags` function returns the control flags of a time base. The following flags are defined (unused flags are set to 0):

`loopTimeBase`
: Indicates whether the time base loops. If this flag is set to 1 and the rate is positive, the time base loops back and restarts from its start time when it reaches its stop time. If this flag is set to 1 and the rate is negative, the time base loops to its stop time. If the flag is set to 0, the movie stops when it reaches the end.

`palindromeLoopTimeBase`
: Indicates whether the time base loops in a palindrome fashion. Palindrome looping causes a time base to move alternately forward and backward. If this flag is set to 1, the time base is palindrome looping.

ERROR CODES

None

SEE ALSO

You can set the control flags of a time base by calling the `SetTimeBaseFlags` function, which is described in the previous section.

GetTimeBaseStatus

Your application can retrieve status information from a time base by calling the `GetTimeBaseStatus` function. This status information allows you to determine when the current time of a time base would fall outside of the range of values specified by the start and stop times of the time base. This can happen when a time base relies on a master time base or when its time has reached the stop time.

```
pascal long GetTimeBaseStatus (TimeBase tb,
                               TimeRecord *unpinnedTime);
```

`tb`
: Specifies the time base for this operation. Your application obtains this time base identifier from the `NewTimeBase` function (described on page 2-316).

`unpinnedTime`
: Contains a pointer to a time structure that is to receive the current time of the time base. Note that this time value may be outside the range of values specified by the start and stop times of the time base.

DESCRIPTION

The `GetTimeBaseStatus` function returns flags that indicate whether the returned time value is outside the range of values specified by the start and stop times of the time base. The following flags are defined (unused flags are set to 0):

`timeBaseBeforeStartTime`
: Indicates that the time value represented by the contents of the time structure referred to by the `unpinnedTime` parameter lies before the start time of the time base. The Movie Toolbox sets this flag to 1 if the current time is before the start time of the time base.

`timeBaseAfterStopTime`
: Indicates that the time value represented by the contents of the time structure referred to by the `unpinnedTime` parameter lies after the stop time of the time base. The Movie Toolbox sets this flag to 1 if the current time is after the stop time of the time base.

ERROR CODES

None

Working With Times

The Movie Toolbox provides a number of functions that allow you to work with time structures. This section describes those functions.

All of these functions work with time structures (see “The Time Structure” on page 2-77 for a complete discussion of the time structure). You can use time structures to represent either time values or durations. Time values specify a point in time, relative to a given time base. Durations specify a span of time, relative to a given time scale. Durations are represented by time structures that have the time base set to 0 (that is, the `base` field in the time structure is set to `nil`).

You can use the `ConvertTime` function to convert a time you obtain from one time base into a time that is relative to another time base. Similarly, you can use the `ConvertTimeScale` function to convert a time from one time scale to another.

You can add two times by calling the `AddTime` function; you can subtract two times with the `SubtractTime` function.

AddTime

The `AddTime` function adds two times. You must specify the times in time structures.

```
pascal void AddTime (TimeRecord *dst, const TimeRecord *src);
```

`dst`
: Contains a pointer to a time structure. This time structure contains one of the operands for the addition. The `AddTime` function returns the result of the addition into this time structure.

`src` Contains a pointer to a time structure. The Movie Toolbox adds this value to the time or duration specified by the `dst` parameter.

DESCRIPTION

If these times are relative to different time scales or time bases, the `AddTime` function converts the times as appropriate to yield reasonable results. However, the time bases for both time values must rely on the same time source.

The result value is formatted based on the operands as follows:

`dst`	`src`	Result
Duration	Duration	Duration
Time value	Duration	Time value

ERROR CODES

None

SubtractTime

The `SubtractTime` function subtracts one time from another. You must specify the times in time structures.

```
pascal void SubtractTime (TimeRecord *dst, const TimeRecord *src);
```

`dst` Contains a pointer to a time structure. This time structure contains one of the operands for the subtraction. The `SubtractTime` function returns the result of the subtraction into this time structure.

`src` Contains a pointer to a time structure. The Movie Toolbox subtracts this value from the time or duration specified by the `dst` parameter.

DESCRIPTION

If these times are relative to different time scales or time bases, the `SubtractTime` function converts the times as appropriate to yield reasonable results. However, the time bases for both time values must rely on the same time source.

The result value is formatted based on the operands as follows:

`dst`	`src`	Result
Time value	Duration	Duration
Duration	Duration	Duration
Time value	Time value	Duration

ERROR CODES

None

ConvertTime

You can convert a time you obtain from one time base into a time that is relative to another time base by calling the `ConvertTime` function. Both time bases must rely on the same time source. You must specify the time to be converted in a time structure.

```
pascal void ConvertTime (TimeRecord *inout, TimeBase newBase);
```

`inout` Contains a pointer to a time structure that contains the time value to be converted. The `ConvertTime` function replaces the contents of this time structure with the time value relative to the specified time base.

`newBase` Specifies the time base for this operation. Your application obtains this time base identifier from the `NewTimeBase` function (described on page 2-316).

DESCRIPTION

The `ConvertTime` function includes the rate associated with each time value in the conversion; therefore, you should use this function when you want to convert time values. Use the `ConvertTimeScale` function (described in the next section) to convert durations.

ERROR CODES

None

ConvertTimeScale

You can convert a time from one time scale into a time that is relative to another time base by calling the `ConvertTimeScale` function. You must specify the time to be converted in a time structure.

```
pascal void ConvertTimeScale (TimeRecord *inout,
                              TimeScale newScale);
```

`inout` Contains a pointer to a time structure that contains the time value to be converted. The `ConvertTimeScale` function replaces the contents of this time structure with the time value relative to the specified time scale.

`newScale` Specifies the time scale for this operation.

DESCRIPTION

The `ConvertTimeScale` function does not include the rate associated with the time value in the conversion; therefore, you should use this function when you want to convert time durations, but not when converting time values. Use the `ConvertTime` function (described in the previous section) to convert time values.

ERROR CODES

None

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—but 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.

NewCallBack

The NewCallback function creates a new callback event. The callback event created at this time is not active until you schedule it by calling the CallMeWhen function, which is described in the next section.

S **WARNING**
You must not call this function at interrupt time. s

```
pascal QTCallBack NewCallBack (TimeBase tb, short cbType);
```

tb Specifies the callback event's time base. You obtain this identifier from the NewTimeBase function (described on page 2-316).

cbType Specifies when the callback event is to be invoked. The value of this field governs how the Movie Toolbox interprets the data supplied in the param1, param2, and param3 parameters to the CallMeWhen function, which is described in the next section. The following values are valid for this parameter:

callBackAtTime
: Indicates that the event is to be invoked at a specified time.

callBackAtRate
: Indicates that the event is to be invoked when the rate for the time base reaches a specified value.

callBackAtTimeJump
: Indicates that the event is to be invoked when the time base's time value changes by an amount that differs from its rate.

callBackAtExtremes
: Indicates that the event is to be invoked when the time base reaches its start time or its stop time. If the start or stop time of the time base changes, the call back is automatically rescheduled. This is very useful for looping or determining when a movie is complete. You determine when the callback is to be fired with the triggerAtStart and triggerAtStop constants. Both flags may be set.

In addition, if the high-order bit of the cbType parameter is set to 1 (this bit is defined by the callBackAtInterrupt flag), the event can be invoked at interrupt time.

DESCRIPTION

The NewCallBack function returns a reference to the new callback event. You must provide this reference to other Movie Toolbox functions described in this section. If the Movie Toolbox cannot create the callback event, this function returns nil.

ERROR CODES

None

CallMeWhen

You schedule a callback event by calling the `CallMeWhen` function. You can call this function from your callback function.

```
pascal OSErr CallMeWhen (QTCallBack cb,
                         QTCallBackProc callBackProc,
                         long refcon, long param1,
                         long param2, long param3);
```

`cb` Specifies the callback event for the operation. You obtain this identifier from the `NewCallBack` function, which is described in the previous section.

`callBackProc`
Points to your callback function.

Your callback function must have the following form:

```
pascal void MyCallBackProc (QTCallBack cb,
                            long refcon);
```

See “Callback Event Functions” on page 2-364 for details.

`refcon` Contains a reference constant value for your callback function.

`param1` Contains scheduling information. The Movie Toolbox interprets this parameter based on the value of the `cbType` parameter to the `NewCallBack` function, described in the previous section.

If `cbType` is set to `callBackAtTime`, the `param1` parameter contains flags indicating when to invoke your callback function for this callback event. The following values are defined (be sure to set unused flags to 0):

`triggerTimeFwd`
Indicates that your callback function should be called at the time specified by `param2` only when time is moving forward (positive rate). The value of this flag is 0x0001.

`triggerTimeBwd`
Indicates that your callback function should be called at the time specified by `param2` only when time is moving backward (negative rate). The value of this flag is 0x0002.

`triggerTimeEither`
Indicates that your callback function should be called at the time specified by `param2` without regard to direction, but the rate must be nonzero. The value of this flag is 0x0003.

If the cbType parameter is set to callBackAtRate, param1 contains flags indicating when to invoke your callback function for this event. The following values are defined (be sure to set unused flags to 0):

triggerRateChange
: Indicates that your callback function should be called whenever the rate changes. The value of this flag is 0x0000.

triggerRateLT
: Indicates that your callback function should be called when the rate changes to a value less than that specified by param2. The value of this flag is 0x0004.

triggerRateGT
: Indicates that your callback function should be called when the rate changes to a value greater than that specified by param2. The value of this flag is 0x0008.

triggerRateEqual
: Indicates that your callback function should be called when the rate changes to a value equal to that specified by param2. The value of this flag is 0x0010.

triggerRateLTE
: Indicates that your callback function should be called when the rate changes to a value that is less than or equal to that specified by param2. The value of this flag is 0x0014.

triggerRateGTE
: Indicates that your callback function should be called when the rate changes to a value that is less than or equal to that specified by param2. The value of this flag is 0x0018.

triggerRateNotEqual
: Indicates that your callback function should be called when the rate changes to a value that is not equal to that specified by param2. The value of this flag is 0x001C.

param2 Contains scheduling information. The Movie Toolbox interprets this parameter based on the value of the cbType parameter to the NewCallBack function, described in the previous section.

If cbType is set to callBackAtTime, the param2 parameter contains the time value at which your callback function is to be invoked for this event. The param1 parameter contains flags affecting when the Movie Toolbox calls your function.

If cbType is set to callBackAtRate, the param2 parameter contains the rate value at which your callback function is to be invoked for this event. The param1 parameter contains flags affecting when the Movie Toolbox calls your function.

param3 Contains the time scale in which to interpret the time value that is stored in param3 if cbType is set to callBackAtTime.

ERROR CODES

None

CancelCallBack

You use the `CancelCallBack` function to cancel a callback event before it executes.

```
pascal void CancelCallBack (QTCallBack cb);
```

`cb` Specifies the callback event for this operation. You obtain this value from the `NewCallBack` function (described on page 2-336).

DESCRIPTION

The `CancelCallBack` function removes the callback event from the list of callback events maintained by the Movie Toolbox. The Movie Toolbox calls this function automatically when it invokes your callback function. In order for a callback event to be scheduled, you must call the `CallMeWhen` function, which is described in the previous section.

ERROR CODES

None

DisposeCallBack

The `DisposeCallBack` function disposes of the memory associated with the specified callback event and cancels the event if it is pending. You should call this function when you are done with each callback event.

s **WARNING**
You must not call this function at interrupt time. s

```
pascal void DisposeCallBack (QTCallBack cb);
```

`cb` Specifies the callback event for the operation. You obtain this value from the `NewCallBack` function (described on page 2-336).

ERROR CODES

None

GetCallBackTimeBase

You can retrieve the time base of a callback event by calling the `GetCallBackTimeBase` function. Your application specifies the callback event's time base by calling the `NewCallBack` function, which is described on page 2-336.

```
pascal TimeBase GetCallBackTimeBase (QTCallBack cb);
```

`cb` Specifies the callback event for the operation. You obtain this value from the `NewCallBack` function.

DESCRIPTION

The `GetCallBackTimeBase` function returns a reference to the callback event's time base.

ERROR CODES

None

GetCallBackType

You can retrieve a callback event's type by calling the `GetCallBackType` function. You specify the type value when you call the `NewCallBack` function (described on page 2-336).

```
pascal short GetCallBackType (QTCallBack cb);
```

`cb` Specifies the callback event for the operation. You obtain this value from the `NewCallBack` function.

DESCRIPTION

The `GetCallBackTimeBase` function returns the callback event's type value. The following values are valid:

`callBackAtTime`
Indicates that the event is to be invoked at a specified time.

`callBackAtRate`
Indicates that the event is to be invoked when the rate for the time base reaches a specified value.

`callBackAtTimeJump`
: Indicates that the event is to be invoked when the time base's time value changes by an amount that differs from its rate.

In addition, if the high-order bit of the returned value is set to 1 (this bit is defined by the `callBackAtInterrupt` flag), the event can be invoked at interrupt time.

ERROR CODES

None

Matrix Functions

The Movie Toolbox provides a number of functions that allow you to work with transformation matrices. This section describes those functions. For more information about transformation matrices, see "The Transformation Matrix" on page 2-26. For descriptions of fixed-point and fixed-rectangle structures, see "The Fixed-Point and Fixed-Rectangle Structures" on page 2-78.

Note

The functions described in this section do not appear in the MPW interface file `Movies.h`; rather, they appear in the `ImageCompression.h` interface file. u

SetIdentityMatrix

The `SetIdentityMatrix` function allows your application to set the contents of a matrix so that it performs no transformation. Such matrices are referred to as *identity matrices*.

```
pascal void SetIdentityMatrix (MatrixRecord *matrix);
```

`matrix`
: Contains a pointer to a matrix structure. The `SetIdentityMatrix` function updates the contents of this matrix so that the matrix describes the identity matrix.

ERROR CODES

None

GetMatrixType

The `GetMatrixType` function allows your application to obtain information about a matrix. This information indicates the nature of the transformation defined by the matrix.

```
pascal short GetMatrixType (MatrixRecordPtr m);
```

`m` Points to the matrix for this operation.

DESCRIPTION

The `GetMatrixType` function returns an integer that indicates the nature of the transformation defined by the matrix. The following values are possible:

`identityMatrixType`
Indicates that the specified matrix is an identity matrix.

`translateMatrixType`
Indicates that the specified matrix defines a translation operation.

`scaleMatrixType`
Indicates that the specified matrix defines a scaling operation.

`scaleTranslateMatrixType`
Indicates that the specified matrix defines both a translation operation and a scaling operation.

`linearMatrixType`
Indicates that the specified matrix defines a rotation, skew, or shear operation.

`linearTranslateMatrixType`
Indicates that the specified matrix defines both a translation operation and a rotation, skew, or shear operation.

`perspectiveMatrixType`
Indicates that the specified matrix defines a perspective (nonlinear) operation.

ERROR CODES

None

CopyMatrix

The `CopyMatrix` function copies the contents of one matrix into another matrix.

```
pascal void CopyMatrix (MatrixRecordPtr m1, MatrixRecord *m2);
```

`m1` Specifies the source matrix for the copy operation.

`m2` Contains a pointer to the destination matrix for the copy operation. The `CopyMatrix` function copies the values from the matrix specified by the `m1` parameter into this matrix.

DESCRIPTION

The `CopyMatrix` function is a convenience function for copying the contents of one matrix to another. You can achieve the same results by using the Memory Manager's `BlockMove` routine, or by assigning the contents of one matrix record to another directly.

ERROR CODES

None

EqualMatrix

The `EqualMatrix` function compares two matrices and returns a result that indicates whether the matrices are equal.

```
pascal Boolean EqualMatrix (const MatrixRecord *m1,
                            const MatrixRecord *m2);
```

`m1` Contains a pointer to one matrix for the compare operation.

`m2` Contains a pointer to the other matrix for the compare operation.

DESCRIPTION

The `EqualMatrix` function returns a Boolean value that indicates whether the specified matrices are equal. If the matrices are equal, the function sets this returned value to `true`. Otherwise, it sets the returned value to `false`.

ERROR CODES

None

TranslateMatrix

The `TranslateMatrix` function allows your application to add a translation value to a specified matrix.

```
pascal void TranslateMatrix (MatrixRecord *m,
                             Fixed deltaH, Fixed deltaV);
```

`m`	Contains a pointer to the matrix structure for this operation.
`deltaH`	Specifies the value to be added to the x coordinate translation value.
`deltaV`	Specifies the value to be added to the y coordinate translation value.

ERROR CODES

None

ScaleMatrix

The `ScaleMatrix` function allows your application to modify the contents of a matrix so that it defines a scaling operation.

```
pascal void ScaleMatrix (MatrixRecord *m, Fixed scaleX,
                         Fixed scaleY, Fixed aboutX, Fixed aboutY);
```

`m`	Contains a pointer to a matrix structure. The `ScaleMatrix` function updates the contents of this matrix so that the matrix describes a scaling operation—that is, it concatenates the respective transformations onto whatever was initially in the matrix structure. You specify the magnitude of the scaling operation with the `scaleX` and `scaleY` parameters. You specify the anchor point with the `aboutX` and `aboutY` parameters.
`scaleX`	Specifies the scaling factor applied to x coordinates.
`scaleY`	Specifies the scaling factor applied to y coordinates.
`aboutX`	Specifies the x coordinate of the anchor point.
`aboutY`	Specifies the y coordinate of the anchor point.

ERROR CODES

None

RotateMatrix

The `RotateMatrix` function allows your application to modify the contents of a matrix so that it defines a rotation operation.

```
pascal void RotateMatrix (MatrixRecord *m, Fixed degrees,
                          Fixed aboutX, Fixed aboutY);
```

`m`	Contains a pointer to a matrix structure. The `RotateMatrix` function updates the contents of this matrix so that the matrix describes a rotation operation—that is, it concatenates the rotation transformations onto whatever was initially in the matrix structure. You specify the direction and amount of rotation with the `degrees` parameter. You specify the point of rotation with the `aboutX` and `aboutY` parameters.
`degrees`	Specifies the number of degrees of rotation.
`aboutX`	Specifies the x coordinate of the anchor point of rotation.
`aboutY`	Specifies the y coordinate of the anchor point of rotation.

ERROR CODES

None

SkewMatrix

The `SkewMatrix` function allows your application to modify the contents of a matrix so that it defines a skew transformation. A skew operation alters the display of an element along one dimension—for example, converting a rectangle into a parallelogram is a skew operation.

```
pascal void SkewMatrix (MatrixRecord *m, Fixed skewX, Fixed skewY,
                        Fixed aboutX, Fixed aboutY);
```

`m`	Contains a pointer to the matrix for this operation. The `SkewMatrix` function updates the contents of this matrix so that it defines a skew operation—that is, it concatenates the respective transformations onto whatever was initially in the matrix structure. You specify the magnitude and direction of the skew operation with the `skewX` and `skewY` parameters. You specify an anchor point with the `aboutX` and `aboutY` parameters.
`skewX`	Specifies the skew value to be applied to x coordinates.

skewY	Specifies the skew value to be applied to y coordinates.
aboutX	Specifies the x coordinate of the anchor point.
aboutY	Specifies the y coordinate of the anchor point.

ConcatMatrix

The ConcatMatrix function concatenates two matrices, combining the transformations described by both matrices into a single matrix.

```
pascal void ConcatMatrix (MatrixRecord *a, MatrixRecord *b);
```

a	Contains a pointer to the source matrix.
b	Contains a pointer to the destination matrix. The ConcatMatrix function performs a matrix multiplication operation, combining the two matrices, and leaves the result in the matrix specified by this parameter.

DESCRIPTION

The form of the operation that the ConcatMatrix function performs is shown by the following formula:

[B] = [B] x [A]

This is a matrix multiplication operation. Note that matrix multiplication is not commutative.

ERROR CODES

None

InverseMatrix

The InverseMatrix function creates a new matrix that is the inverse of a specified matrix.

```
pascal Boolean InverseMatrix (MatrixRecord *m,
                              MatrixRecord *im);
```

`m`	Contains a pointer to the source matrix for the operation.
`im`	Contains a pointer to a matrix structure that is to receive the new matrix. The `InverseMatrix` function updates this structure so that it contains a matrix that is the inverse of that specified by the `m` parameter.

DESCRIPTION

The `InverseMatrix` function returns a Boolean value that indicates whether it could create an inverse matrix. If the function could create an inverse matrix, it sets this returned value to `true`. Otherwise, the function sets the returned value to `false`.

ERROR CODES

None

TransformPoints

The `TransformPoints` function allows your application to transform a set of QuickDraw points through a specified matrix.

```
pascal OSErr TransformPoints (MatrixRecord *mp, Point *pt1,
                              long count);
```

`mp`	Contains a pointer to the transformation matrix for this operation.
`pt1`	Contains a pointer to the first QuickDraw point to be transformed.
`count`	Specifies the number of QuickDraw points to be transformed. These points must be stored immediately following the point specified by the `pt1` parameter.

ERROR CODES

None

SEE ALSO

You can transform a set of QuickDraw points that are made up of fixed values by calling the `TransformFixedPoints` function, which is described in the next section.

TransformFixedPoints

The `TransformFixedPoints` function allows your application to transform a set of fixed points through a specified matrix.

```
pascal OSErr TransformFixedPoints (MatrixRecord *m,
                                   FixedPoint *fpt, long count);
```

`m`	Contains a pointer to the transformation matrix for this operation.
`fpt`	Contains a pointer to the first fixed point to be transformed.
`count`	Specifies the number of fixed points to be transformed. These points must be stored immediately following the point specified by the `fpt` parameter.

ERROR CODES

None

SEE ALSO

You can transform a set of fixed points that is made up of short integer values by calling the `TransformPoints` function, which is described in the previous section.

TransformRect

The `TransformRect` function allows your application to transform the upper-left and lower-right points of a rectangle through a specified matrix.

```
pascal Boolean TransformRect (MatrixRecordPtr m, Rect *r,
                              FixedPoint *fpp);
```

`m`	Specifies the matrix for this operation.
`r`	Contains a pointer to the structure that defines the rectangle to be transformed. The `TransformRect` function returns the updated coordinates into the structure referred to by this parameter. If the resulting rectangle has been rotated or skewed (that is, the transformation involves operations other than scaling and translation), the function sets the returned Boolean value to `false` and returns the coordinates of the rectangle that encloses the transformed rectangle. The function then updates the points specified by the `fpp` parameter to contain the coordinates of the four corners of the transformed rectangle.

`fpp` Contains a pointer to an array of four fixed points. The `TransformRect` function returns the coordinates of the four corners of the rectangle after the transformation operation.

If you do not want this information, set this parameter to `nil`.

DESCRIPTION

The `TransformRect` function returns a Boolean value indicating the nature of the result rectangle. If the matrix defines transformations other than translation and scaling, the `TransformRect` function sets the returned value to `false`, updates the rectangle specified by the `r` parameter to define the boundary box of the resulting rectangle, and places the coordinates of the corners of the resulting rectangle in the points specified by the `fpp` parameter. If the transformed rectangle and its boundary box are the same, the function sets the returned value to `true`.

ERROR CODES

None

TransformFixedRect

The `TransformFixedRect` function allows your application to transform the upper-left and lower-right points of a rectangle through a specified matrix. This rectangle must be specified by fixed points.

```
pascal Boolean TransformFixedRect (MatrixRecord *m,
                                   FixedRect *fr,
                                   FixedPoint *fpp);
```

`m` Contains a pointer to the matrix for this operation.

`fr` Contains a pointer to the structure that defines the rectangle to be transformed. The `TransformFixedRect` function returns the updated coordinates into the structure referred to by this parameter. If the resulting rectangle has been rotated or skewed (that is, the transformation involves operations other than scaling and translation), the function sets the returned Boolean value to `false` and returns the coordinates of the boundary box of the transformed rectangle. The function then updates the points specified by the `fpp` parameter to contain the coordinates of the four corners of the transformed rectangle.

`fpp` Contains a pointer to an array of four fixed points. The `TransformFixedRect` function returns the coordinates of the four corners of the rectangle after the transformation operation.

If you do not want this information, set this parameter to `nil`.

DESCRIPTION

The `TransformFixedRect` function returns a Boolean value indicating the nature of the result rectangle. If the matrix defines transformations other than translation and scaling, the `TransformFixedRect` function sets the returned value to `false`, updates the rectangle specified by the `fr` parameter to define the boundary box of the resulting rectangle, and places the coordinates of the corners of the resulting rectangle in the points specified by the `fpp` parameter. If the transformed rectangle and its boundary box are the same, the function sets the returned value to `true`.

ERROR CODES

None

SEE ALSO

You can transform a standard rectangle by calling the `TransformRect` function, which is described in the previous section.

TransformRgn

The `TransformRgn` function allows your application to apply a specified matrix to a region.

```
pascal OSErr TransformRgn (MatrixRecordPtr mp, RgnHandle r);
```

`mp`	Points to the matrix for this operation. The `TransformRgn` function currently supports only translating and scaling operations.
`r`	Specifies the region to be transformed. The `TransformRgn` function transforms each point in the region according to the contents of the specified matrix.

ERROR CODES

Memory Manager errors

RectMatrix

The `RectMatrix` function allows your application to create a matrix that performs a translate and scale operation as described by the relationship between two rectangles.

```
pascal void RectMatrix (MatrixRecord *matrix, Rect *srcRect,
                        Rect *dstRect);
```

`matrix` Contains a pointer to a matrix structure. The `RectMatrix` function updates the contents of this matrix so that the matrix describes a transformation from points in the rectangle specified by the `srcRect` parameter to points in the rectangle specified by the `dstRect` parameter. The previous contents of the matrix are ignored.

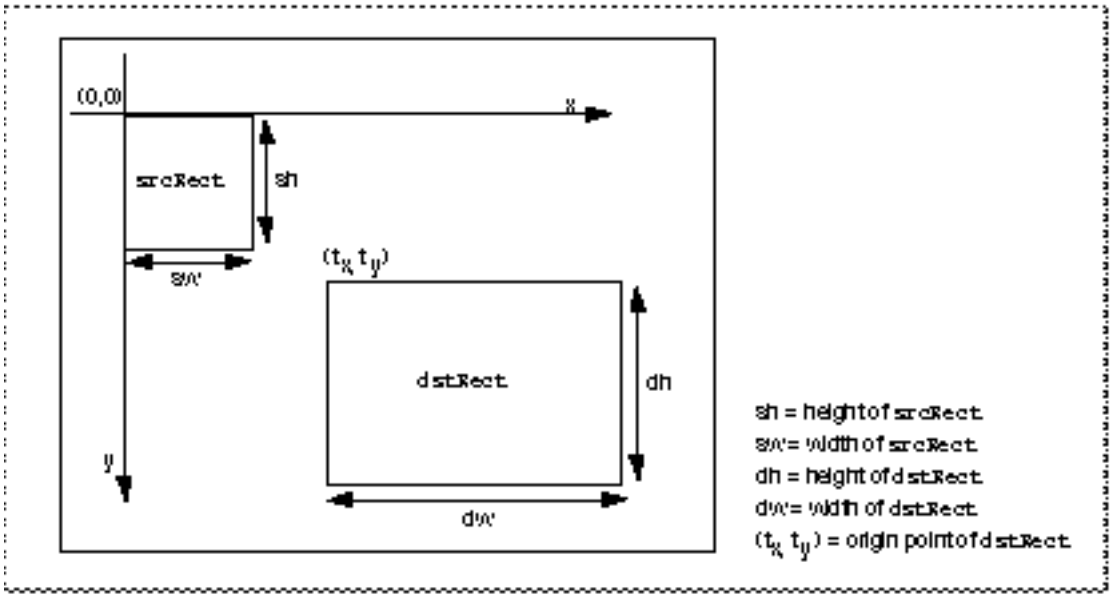
`srcRect` Contains a pointer to the source rectangle.

`dstRect` Contains a pointer to the destination rectangle.

DESCRIPTION

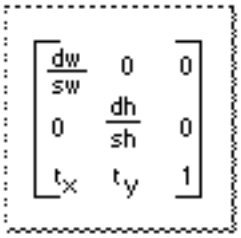
You specify the two rectangles; the function returns the appropriate matrix. Figure 2-43 shows how this matrix transforms the source image.

Figure 2-43 Transforming an image with the `RectMatrix` function



Calling the `RectMatrix` function with the two rectangles shown in Figure 2-43 results in the matrix shown in Figure 2-44.

Figure 2-44 Matrix created as a result of calling the `RectMatrix` function



SEE ALSO

If you call the `TransformRect` function (described on page 2-348) and supply the matrix produced by the `RectMatrix` function along with the source rectangle you specified when you called the `RectMatrix` function, the result is equivalent to the destination rectangle you specified.

MapMatrix

The `MapMatrix` function alters an existing matrix so that it defines a transformation from one rectangle to another, similar to the `MapRect` and `MapRegion` routines that are described in *Inside Macintosh: Imaging*.

```
pascal void MapMatrix (MatrixRecord *matrix, Rect *fromRect,
                       Rect *toRect);
```

`matrix` Contains a pointer to a matrix structure. The `MapMatrix` function modifies this matrix so that it performs a transformation in the rectangle specified by the `toRect` parameter that is analogous to the transformation it currently performs in the rectangle specified by the `fromRect` parameter.

`fromRect` Contains a pointer to the source rectangle.

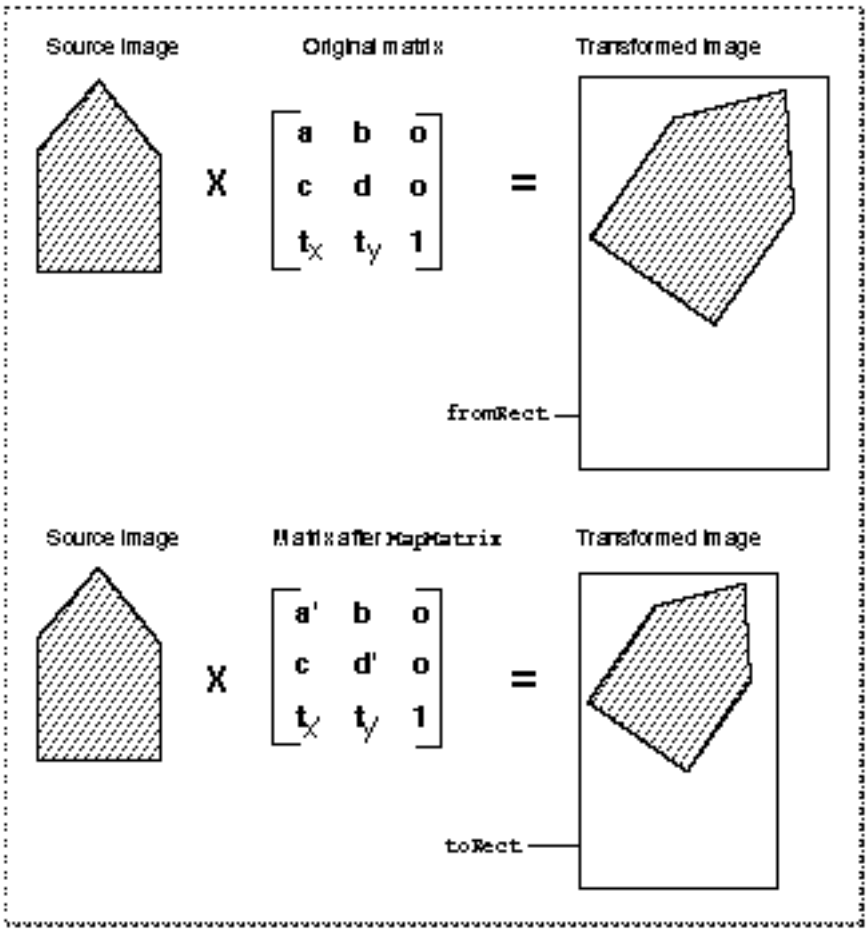
`toRect` Contains a pointer to the destination rectangle.

DESCRIPTION

The `MapMatrix` function affects only the scaling and translation attributes of the matrix. This function is similar to `RectMatrix`, with the exception that `MapMatrix` concatenates the translation and scaling operations to the previous contents of the matrix, whereas `RectMatrix` first sets the matrix to the identity state.

Figure 2-45 shows how the matrix that you obtain from the `MapMatrix` function transforms a source image.

Figure 2-45 Transforming an image with the `MapMatrix` function



SEE ALSO

You can create a matrix that maps one rectangle to another by calling the `RectMatrix` function, which is described in the previous section.

Application-Defined Functions

This section describes the application-defined functions used with the Movie Toolbox. It is divided into the following topics:

- "Progress Functions" describes the functions that your application must assign to monitor the progress of the Movie Toolbox during long operations
- "Cover Functions" describes the functions that your application must use to perform custom processing whenever one of your movies covers a screen region or reveals a region that was previously covered
- "Error-Notification Functions" discusses the functions that your application must use to perform custom error-processing; you'll find these functions particularly helpful when you're debugging your program
- "Movie Callout Functions"describes the application-defined functions that the Movie Toolbox calls repeatedly while a movie preview is playing; you can use your movie callout function to stop the preview
- "File Filter Functions" provides details about the function that you can supply to filter the files that are displayed to the user in a dialog box
- "Custom Dialog Functions" supplies information about a function that lets you support the template in the custom dialog template that you specified with the `CustomGetFilePreview` function
- "Modal-Dialog Filter Functions" describes the functions that you can provide to support the custom dialog template you specified with your custom dialog function; your modal-dialog filter function gives you greater control over the interface presented to the user
- "Standard File Activation Functions" describes the functions that control the highlighting of any items whose shape is known only by your application
- "Callback Event Functions" discusses the callback events that you can ask the `CallMeWhen` function to schedule
- "Text Functions" describes a function through which you can specify operations on text and whether you want to display the text

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 (described on page 2-155). 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.

MyProgressProc

Your progress function should support the following interface:

```
pascal OSErr MyProgressProc (Movie theMovie, short message,
                             short whatOperation,
                             Fixed percentDone, long refcon);
```

`theMovie` Specifies the movie for this operation. The Movie Toolbox sets this parameter to identify the appropriate movie.

`message` Indicates why the Movie Toolbox called your function. The following values are valid:

`movieProgressOpen`
Indicates the start of a long operation. This is always the first message sent to your function. Your function can use this message to trigger the display of your progress window.

`movieProgressUpdatePercent`
Passes completion information to your function. The Movie Toolbox repeatedly sends this message to your function. The `percentDone` parameter indicates the relative completion of the operation. You can use this value to update your progress window.

`movieProgressClose`
Indicates the end of a long operation. This is always the last message sent to your function. Your function can use this message as an indication to remove its progress window.

`whatOperation`
Indicates the long operation that is currently underway. The following values are valid:

`progressOpFlatten`
Your application has called the `FlattenMovie` or `FlattenMovieData` function (described on page 2-105 and page 2-107, respectively).

progressOpInsertTrackSegment
: Your application has called the InsertTrackSegment function (described on page 2-262). The Movie Toolbox calls the progress function that is assigned to the movie that contains the destination track.

progressOpInsertMovieSegment
: Your application has called the InsertMovieSegment function (described on page 2-257). The Movie Toolbox calls the progress function that is assigned to the destination movie.

progressOpPaste
: Your application has called the PasteMovieSelection function (described on page 2-249). The Movie Toolbox calls the progress function that is assigned to the destination movie.

progressOpAddMovieSelection
: Your application has called the AddMovieSelection function (described on page 2-250). The Movie Toolbox calls the progress function that is assigned to the destination movie. The Movie Toolbox calls the progress function that is assigned to the destination movie.

progressOpCopy
: Your application has called the CopyMovieSelection function (described on page 2-248). The Movie Toolbox calls the progress function that is assigned to the destination movie.

progressOpCut
: Your application has called the CutMovieSelection function (described on page 2-247). The Movie Toolbox calls the progress function that is assigned to the destination movie.

progressOpLoadMovieIntoRam
: Your application has called the LoadMovieIntoRam function (described on page 2-140). The Movie Toolbox calls the progress function that is assigned to the destination movie.

progressOpLoadTrackIntoRam
: Your application has called the LoadTrackIntoRam function (described on page 2-142). The Movie Toolbox calls the progress function that is assigned to the destination track.

progressOpLoadMediaIntoRam
: Your application has called the LoadMediaIntoRam function (described on page 2-143). The Movie Toolbox calls the progress function that is assigned to the destination media.

progressOpImportMovie
: Your application has called the ConvertFileToMovieFile function (described on page 2-93). The Movie Toolbox calls the progress function that is associated with the destination movie file. This flag is also used, as appropriate, for the PasteHandleIntoMovie functions (described on page 2-252).

progressOpExportMovie
: Your application has called the ConvertMovieFile function (described on page 2-95). The Movie Toolbox calls the progress function that is associated with the destination movie. This flag is also used, as appropriate, for the PutMovieIntoTypedHandle function (described on page 2-253).

percentDone
: Contains a fixed-point value indicating how far the operation has progressed. Its value is always between 0.0 and 1.0. This parameter is valid only when the message field is set to moviePrograssUpdatePercent.

refcon
: Reference constant value for use by your progress function. Your application specifies the value of this reference constant when you assign the progress function to the movie.

DESCRIPTION

Your progress function should return an error value. The Movie Toolbox examines this value after each moviePrograssUpdatePercent message and before continuing the current operation. Set this value to a nonzero value, such as userCanceledErr, to cancel the operation; set it to noErr to continue.

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 (described on page 2-159). When your movie covers a region, the default function that is called does nothing.

Use the `SetMovieCoverProcs` function (described on page 2-156) to set both types of cover functions.

MyCoverProc

Your cover functions should support the following interface:

```
pascal OSErr MyCoverProc (Movie theMovie, RgnHandle changedRgn,
                          long refcon);
```

`theMovie` Specifies the movie for this operation.

`changedRgn` Contains a handle to the changed screen region.

`refcon` Contains the reference constant that you specified when you defined the progress function.

DESCRIPTION

Your cover function should always return an error value of `noErr`.

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 (described on page 2-86) allows you to identify your application's error-notification function in the `errProc` parameter.

MyErrProc

The entry point to your error-notification function should take the following form:

```
pascal void MyErrProc (OSErr theErr, long refcon);
```

`theErr` Contains the result code that the Movie Toolbox is going to place in the current error value.

`refcon` Contains the reference constant value that you specified when your application called the `SetMoviesErrorProc` function.

Movie Callout Functions

The `PlayMoviePreview` function (described on page 2-120) 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`.

MyCalloutProc

Your movie callout function should present the following interface:

```
pascal Boolean MyCallOutProc (long refcon);
```

`refcon` Contains the reference constant that you specified when you called the `PlayMoviePreview` function.

DESCRIPTION

Your movie callout function returns a Boolean value. The Movie Toolbox examines this value before continuing. If your function sets this value to `false`, the Movie Toolbox stops the preview and returns to your application.

Note

If you call the `GetMovieActiveSegment` function (described on page 2-137) from within your movie callout function, the Movie Toolbox will have changed the active movie segment to be the preview segment of the movie. The Movie Toolbox restores the active segment when the preview is done playing. u

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 (see *Inside Macintosh: Files* for more information).

MyFileFilter

A file filter function whose address is passed to `SFGetFilePreview`, `StandardGetFilePreview`, or `CustomGetFilePreview` should have the following form.

```
pascal Boolean MyFileFilter (ParmBlkPtr parmBlock);
```

`paramBlock` A pointer to the parameter block associated with the files that are displayed to the user in this dialog box. For details, see *Inside Macintosh: Files*.

DESCRIPTION

When `SFGetFilePreview`, `StandardGetFilePreview`, or `CustomGetFilePreview` is displaying the contents of a volume or folder, it checks the file type of each file and filters out files whose types do not match your application's specifications. If your application also supplies a file filter function, the Standard File Package calls that function each time it identifies a file of an acceptable type.

When your file filter function is called, it is passed, in the `paramBlock` parameter, a pointer to a catalog information parameter block. See the chapter "File Manager" in *Inside Macintosh: Files* for a description of the fields of this parameter block.

Your function evaluates the catalog information parameter block and returns a Boolean value that determines whether the file is filtered (that is, a value of `true` suppresses display of the filename, and a value of `false` allows the display). If you do not supply a file filter function, the Standard File Package displays all files of the specified types.

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. See *Inside Macintosh: Files* for a complete description of the 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. For more information about using custom dialog functions with the CustomGetFile routine, see *Inside Macintosh: Files*.

MyDlgHook

A dialog hook function should have the following form:

```
pascal short MyDlgHook (short item, DialogPtr theDialog,
                        Ptr myDataPtr);
```

item	The number of the item selected.
theDialog	A pointer to the dialog structure for the dialog box.
myDataPtr	A pointer to the optional data whose address is passed to CustomGetFilePreview.

DESCRIPTION

You supply a dialog hook function to handle user selections of items that you added to a dialog box. If you provide a dialog hook function, CustomGetFilePreview calls your function immediately after calling the Dialog Manager's ModalDialog function. It passes your function the item number returned by ModalDialog, a pointer to the dialog structure, and a pointer to the data received from your application, if any.

Your dialog hook function returns as its function result an integer that is either the item number passed to it or some other item number. If your dialog hook function does not handle a selection, it should pass the item number back to the Standard File Package for processing by setting its return value equal to the item number. If your dialog hook function does handle the selection, it should pass back sfHookNullEvent or the number of some other pseudo-item.

SEE ALSO

See *Inside Macintosh: Files* for another sample dialog hook function.

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. See *Inside Macintosh: Files* for more information about using modal-dialog filter functions with `CustomGetFile`.

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. u

MyModalFilter

A modal-dialog filter function whose address is passed to `CustomGetFilePreview` should have the following form:

```
pascal Boolean MyModalFilter (DialogPtr theDialog,
                              EventRecord *theEvent,
                              short itemHit, Ptr myDataPtr);
```

`theDialog`	A pointer to the dialog structure of the dialog box.
`theEvent`	A pointer to the event structure for the event.
`itemHit`	The number of the item selected.
`myDataPtr`	A pointer to the optional data whose address is passed to `CustomGetFilePreview`.

DESCRIPTION

Your modal-dialog filter function determines how the Dialog Manager's `ModalDialog` routine filters events. The `ModalDialog` routine retrieves events by calling the Event Manager's `GetNextEvent` routine. The Standard File Package contains an internal filter function that performs some preliminary processing on each event it receives. If you provide a modal-dialog filter function, `ModalDialog` calls your filter function after it calls the internal Standard File Package filter function and before it sends the event to your dialog hook function.

Your modal-dialog filter function returns a Boolean value that reports whether it handled the event. If your function returns a value of `false`, `ModalDialog` processes the event through its own filters. If your function returns a value of `true`, `ModalDialog` returns with no further action.

SEE ALSO

See *Inside Macintosh: Files* for another sample modal-dialog 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. See *Inside Macintosh: Files* for more information about standard file activation routines.

MyActivateProc

An activation function should have the following form:

```
pascal void MyActivateProc (DialogPtr theDialog, short itemNo,
                            Boolean activating, Ptr myDataPtr);
```

`theDialog` A pointer to the dialog structure of the dialog box.

`itemNo` The number of the item selected.

`activating` A Boolean value that specifies whether the field is being activated (`true`) or deactivated (`false`).

`myDataPtr` A pointer to the optional data whose address is passed to `CustomGetFilePreview`.

DESCRIPTION

Ordinarily, you need to supply an activation function only if your application builds a list from which the user can select entries. The Standard File Package supplies the activation routine for the file display list and for all TextEdit fields. You can also use the activation function to keep track of which field is receiving keyboard input, if your application needs that information.

Your application is responsible for removing the highlighting when one of its fields becomes inactive and for adding the highlighting when one of its fields becomes active. The Standard File Package can handle the highlighting of all TextEdit fields, even those defined by your application.

Callback Event Functions

The `CallMeWhen` function (described on page 2-337) schedules a callback event. You specify the callback event in the `callBackProc` parameter.

MyCallBack

Your callback function must support the following interface:

```
pascal void MyCallBackProc (QTCallBack cb, long refcon);
```

`cb` Specifies the callback event for the operation.

`refcon` Contains a reference constant value for your callback function.

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.

MyTextProc

Your text function should have the following form:

```
pascal OSErr MyTextProc (Handle theText, Movie theMovie,
                         short *displayFlag, long refcon);
```

`theText` Contains a handle to the formatted text.

`theMovie` Specifies the movie for this operation.

`displayFlag`
: Contains a pointer to one of the following flags, which specify how you want the text media handler to proceed when your function returns. The three possible return values for the flag are:

  `txtProcDefaultDisplay`
  : Indicates that the media should follow the instructions of its own `displayFlag` constants.

  `txtProcDontDisplay`
  : Tells the media not to display the text.

  `txtProcDoDisplay`
  : Instructs the media to display the text regardless of the media's own `displayFlag` constants.

`refcon`
: Contains the reference constant to your text function.

Summary of the Movie Toolbox

C Summary

Constants

```
#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 */
};

enum
{
/*
   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
*/
   qtcbNeedsRateChanges       = 1,  /* rate changes */
   qtcbNeedsTimeChanges       = 2,  /* time changes */
   qtcbNeedsStartStopChanges  = 4   /* time base changes at start &
                                       stop times */
};

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

enum
{
/* flags for LoadIntoRAM functions */
   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 */
};

enum
{
/* flag for PasteHandleIntoMovie function */
   pasteInParallel = 1  /* changes function to take contents and type of
                           handle and add to movie */
};

/* text description display flags used in AddTextSample and AddTESample */
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 FindNextText 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;
```

Data Types

```
typedef MovieRecord *Movie;         /* movie identifier */
typedef TrackRecord *Track;         /* track identifer */
typedef MediaRecord *Media;         /* media identifier */
typedef UserDataRecord *UserData;   /* user data list identifier */
typedef TrackEditStateRecord *TrackEditState;
                                    /* track edit state identifier */
typedef MovieEditStateRecord *MovieEditState;
                                    /* movie edit state identifier */
typedef long TimeValue;             /* time value field in time structure */
typedef long TimeScale;             /* time scale field in time structure */
typedef TimeBaseRecord *TimeBase;   /* time base identifier */
typedef CallBackRecord *QTCallBack; /* callback identifier */

typedef Int64 CompTimeValue;

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

```
struct TimeRecord
{
      CompTimeValue  value;/* time value as duration or absolute */
      TimeScale      scale;/* time scale as unit of time & number of units */
      TimeBase       base; /* reference to the time base */
};
typedef struct TimeRecord TimeRecord;

/* All sample descriptions start with this header. */

struct SampleDescription
{
      long     descSize;         /* total size in bytes of this structure */
      long     dataFormat;       /* format of the sample data */
      long     resvd1;           /* reserved--set to 0 */
      short    resvd2;           /* reserved--set to 0 */
      short    dataRefIndex;     /* reserved--set to 1 */
};
typedef struct SampleDescription SampleDescription;
typedef SampleDescription *SampleDescriptionPtr, **SampleDescriptionHandle;

struct SoundDescription
{
      long     descSize;   /* total size in bytes of this structure */
      long     dataFormat; /* format of the sound data */
      long     resvd1;     /* reserved--set to 0 */
      short    resvd2;     /* reserved--set to 0 */
      short    dataRefIndex;
                           /* reserved--set to 1 */
      short    version;    /* which version is this data? */
      short    revlevel;   /* which version of that codec did this? */
      long     vendor;     /* which codec compressed this data? */
      short    numChannels;/* number of sound channels used by sample */
      short    sampleSize; /* number of bits in each sound sample */
      short    compressionID;
                           /* sound compression used--set to 0 if none */
      short    packetSize; /* packet size for compression--set to 0 if
                              no compression */
      Fixed    sampleRate; /* rate at which sound samples were obtained */
};
typedef struct SoundDescription SoundDescription;
typedef SoundDescription *SoundDescriptionPtr, **SoundDescriptionHandle;
```

```
typedef struct TextDescription
{
   long     size;          /* total size of this text description
                               structure */
   long     type;          /* type of data in this structure ('text') */
   long     resvd1;        /* reserved for use by Apple--set to 0 */
   short    resvd2;        /* reserved for use by Apple--set to 0 */
   short    dataRefIndex;  /* index to data references */
   long     displayFlags;  /* display flags for text */
   long     textJustification;
                           /* text justification flags */
   RGBColor bgColor;       /* background color */
   Rect     defaultTextBox;/* location of the text within track bounds */
   ScrpSTElement defaultStyle;
                           /* default style (TextEdit structure) */
} TextDescription, *TextDescriptionPtr, **TextDescriptionHandle;

typedef struct TextDescription TextDescription;
typedef TextDescription *TextDescriptionPtr, **TextDescriptionHandle;

/* pointer to application-defined movie progress function */
typedef pascal OSErr  (*MovieProgressProcPtr) (Movie theMovie, short message,
                        short whatOperation, Fixed percentDone, long refcon);

/* pointer to application-defined cover function */
typedef pascal OSErr (*MovieRgnCoverProc)(Movie theMovie,
                      RgnHandle changedRgn, long refcon);

typedef Handle MediaInformationHandle;  /* data returned by media handler */
typedef ComponentInstance MediaHandler; /* media handler */
typedef Component MediaHandlerComponent; /* media handler component */
typedef ComponentInstance DataHandler;  /* data handler */
typedef Component DataHandlerComponent; /* data handler component */
typedef ComponentResult HandlerError;   /* error handler */
typedef ComponentInstance MovieController;
                                        /* movie controller */

/* pointer to application-defined error-notification function */
typedef pascal void (*ErrorProcPtr)(OSErr theErr, long refcon);

/* pointer to application-defined movie preview callout function */
typedef pascal Boolean (*MoviePreviewCallOutProc)(long refcon);
```

```
enum
{
/* control flags for timeBaseFlags parameter of SetTimeBaseFlags function */
   loopTimeBase               = 1,  /* whether time base loops */
   palindromeLoopTimeBase     = 2   /* whether time base loops in palindrome
                                       fashion */
};

typedef unsigned long TimeBaseFlags;   /* control flags for time base */

/* pointer to application-defined callback function */
typedef pascal void (*QTCallBackProc)(QTCallBack cb, long refcon);

struct QTCallBackHeader
{
      long     callBackFlags;    /* clock component scheduling data flags */
      long     reserved1;        /* reserved for use by Apple */
      char     qtPrivate[40];    /* reserved for use by Apple */
};

struct MatrixRecord
{
      Fixed matrix[3][3];
};
typedef struct FixedPoint FixedPoint;
struct FixedPoint
{
      Fixed x;    /* point's x coordinate as fixed-point number */
      Fixed y;    /* point's y coordinate as fixed-pont number */
};
typedef struct MatrixRecord  MatrixRecord;
typedef MatrixRecord  *MatrixRecordPtr;/* pointer to matrix structure */

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

```
enum
{
/* progress function messages */
   movieProgressOpen          = 0,  /* indicates start of a long operation */
   movieProgressUpdatePercent = 1,  /* passes completion data to function */
   movieProgressClose         = 2   /* indicates end of a long operation */
};
typedef unsigned char movieProgressMessages;

enum
{
/*
   progress function operations that tell which function your application
   has called
*/
   progressOpFlatten                  = 1,  /* FlattenMovie or
                                               FlattenMovieData */
   progressOpInsertTrackSegment       = 2,  /* InsertTrackSegment */
   progressOpInsertMovieSegment       = 3,  /* InsertMovieSegment */
   progressOpPaste                    = 4,  /* PasteMovieSelection */
   progressOpAddMovieSelection        = 5,  /* AddMovieSelection */
   progressOpCopy                     = 6,  /* CopyMovieSelection */
   progressOpCut                      = 7,  /* CutMovieSelection */
   progressOpLoadMovieIntoRam         = 8,  /* LoadMovieIntoRam */
   progressOpLoadTrackIntoRam         = 9,  /* LoadTrackIntoRam */
   progressOpLoadMediaIntoRam         = 10, /* LoadMediaIntoRam */
   progressOpImportMovie              = 11, /* ConvertFileToMovieFile */
   progressOpExportMOvie              = 12  /* ConvertMovieFile */
};
typedef unsigned char movieProgressOperations;

enum
{
/* NewMovie function flags */
   newMovieActive                     = 1<<0,     /* new movie is or
                                                     is not active */
   newMovieDontResolveDataRefs        = 1<<1,     /* data reference
                                                     resolution level */
   newMovieDontAskUnresolvedDataRefs  = 1<<2,     /* is user asked to locate
                                                     files? */
```

```
   newMovieDontAutoAlternates          = 1<<3      /* are enabled tracks
                                                      selected from
                                                      alternate groups? */
};
typedef unsigned char newMovieFlagsEnum;

/* track usage flags in SetTrackUsage function */
enum
{
   trackUsageInMovie       = 1<<1,  /* track is used in movie */
   trackUsageInPreview     = 1<<2,  /* track is used in preview */
   trackUsageInPoster      = 1<<3   /* track is used in poster */
};
typedef unsigned char trackUsageEnum;

/* media sample flags in AddMediaSample function */
enum
{
   mediaSampleNotSync      = 1<<0,  /* sample to be added is not a
                                       sync sample */
   mediaSampleShadowSync   = 1<<1   /* sample is a shadow sync sample */
};
typedef unsigned char mediaSampleFlagsEnum;

enum
{
/*
   interesting times flags in interestingTimeFlags parameter of
   GetMovieNextInterestingTime function
*/
   nextTimeMediaSample  = 1<<0,  /* finds next sample in movie's media */
   nextTimeMediaEdit    = 1<<1,  /* finds next sample group in movie's
                                    media */
   nextTimeTrackEdit    = 1<<2,  /* finds sample for next entry in edit
                                    list */
   nextTimeSyncSample   = 1<<3,  /* finds next sync sample in movie's
                                    media */
   nextTimeEdgeOK       = 1<<14,
                                 /* to receive element data at specified
                                    time */
```

```
   nextTimeIgnoreActiveSegment
                        = 1<<15
                                /* look outside active segment for
                                   samples */
};
typedef unsigned short nextTimeFlagsEnum;

enum
{
/* movie-creation flags from CreateMovieFile function */
   createMovieFileDeleteCurFile     = 1L<<31,   /* delete existing file? */
   createMovieFileDontCreateMovie   = 1L<<30,   /* is new movie created? */
   createMovieFileDontOpenFile      = 1L<<29    /* is new movie file
                                                   opened? */
};
typedef unsigned long createMovieFileFlagsEnum;

/* movie-flattening flags from FlattenFlags function */
enum
{
   flattenAddMovieToDataFork     = 1L<<0,/* movie placed in data fork */
   flattenActiveTracksOnly       = 1L<<2,/* enabled movie tracks added */
   flattenDontInterleaveFlatten  = 1L<<3 /* disable data storage
                                            optimizations */
};
typedef unsigned long movieFlattenFlagsEnum;

enum
{
/* movie scrap flags from PutMovieOnScrap function */
   movieScrapDontZeroScrap= 1<<0,/* is scrap cleared before movie is
                                    put on scrap? */
   movieScrapOnlyPutMovie = 1<<1 /* are other items placed on scrap along
                                    with movie? */
};
typedef unsigned char movieScrapFlagsEnum;
```

```
enum
{
/*
   callback flags from CallMeWhen function specify when callback
   should be called
*/
   triggerTimeFwd       = 0x0001, /* only when time is at positive rate */
   triggerTimeBwd       = 0x0002, /* only when time is at negative rate */
   triggerTimeEither    = 0x0003, /* at specified time without regard
                                     to rate */
   triggerRateLT        = 0x0004, /* whenever rate changes */
   triggerRateGT        = 0x0008, /* when changed rate greater than param2 */
   triggerRateEqual     = 0x0010, /* when changed rate equal to param2 */
   triggerRateLTE       = triggerRateLT | triggerRateEqual,
                             /* when rate less than or equal to
                                param2 */
   triggerRateGTE       = triggerRateGT | triggerRateEqual,
                            /* when rate greater than or equal to
                               param2 */
   triggerRateNotEqual  = triggerRateGT | triggerRateEqual | triggerRateLT,
                            /* when rate is not equal to param2 */
   triggerRateChange    = 0      /* whenever rate changes */
   triggerAtStart       = 0x0001,
                            /* at startup time */
   triggerAtStop        = 0x0002 /* at stop time */
};
typedef unsigned short QTCallBackFlags;

enum
{
/*
   flags returned by GetTimeBaseStatus function specify where time value in
   time structure lies
*/
   timeBaseBeforeStartTime    = 1,  /* before start time of time base */
   timeBaseAfterStopTime      = 2   /* after stop time of time base */
};
typedef unsigned long TimeBaseStatus;
```

```
enum
{
/*
   values for cbType parameter of NewCallBack function specify when event
   can be invoked
*/
   callBackAtTime       = 1,        /* at a specified time */
   callBackAtRate       = 2,        /* time base rate at specified value */
   callBackAtTimeJump   = 3,        /* time value jumps unexpectedly */
   callBackAtExtremes   = 4         /* time value at start time, stop time,
                                       or either */
   callBackAtInterrupt  = 0x8000    /* at interrupt time */
};
typedef unsigned short QTCallBackType;

enum
{
   identityMatrixType         = 0x00,  /* identity matrix */
   translateMatrixType        = 0x01,  /* translation operation */
   scaleTranslateMatrixType   = 0x03,  /* translation & scaling operation */
   linearMatrixType           = 0x04,  /* rotation, skew, or shear
                                          operation */
   linearTranslateMatrixType  = 0x05,  /* translation & rotation, skew,
                                          or shear operation */
   perspectiveMatrixType      = 0x06   /* perspective (nonlinear)
                                          operation */
};
typedef unsigned short MatrixFlags;

enum
{
/*
   values for the dataRefAttributes parameter of the GetMediaDataRef
   function
*/
   dataRefSelfReference  = 1<<0,/* is reference to movie resource's
                                   data file? */
   dataRefWasNotResolved = 1<<1/* did Movie Toolbox resolve reference? */
};
typedef unsigned long dataRefAttributesFlags;
```

```
enum
{
/* flags for SetMoviePlayHints and SetMediaPlayHints functions */
   hintsScrubMode       = 1<<0,  /* mask == && (if flags == scrub on,
                                    flags != scrub off) */
   hintsAllowInterlace  = 1<<6,
   hintsUseSoundInterp  = 1<<7
} ;
typedef unsigned long playHintsEnum;

enum
{
   mediaHandlerFlagGenericClient = 1   /* component flag--should be set for
                                          all media handler components that
                                          make use of generic media
                                          handler */
};
typedef unsigned long mediaHandlerFlagsEnum;
```

Functions for Getting and Playing Movies

Initializing the Movie Toolbox

```
pascal OSErr EnterMovies    (void);
pascal void ExitMovies      (void);
```

Error Functions

```
pascal OSErr GetMoviesError
                            (void);
pascal OSErr GetMoviesStickyError
                            (void);
pascal void ClearMoviesStickyError
                            (void);
pascal void SetMoviesErrorProc
                            (ErrorProcPtr errProc, long refcon);
```

Movie Functions

```
pascal OSErr NewMovieFromFile
                            (Movie *theMovie, short resRefNum,
                             short *resId, StringPtr resName,
                             short newMovieFlags,
                             Boolean *dataRefWasChanged);
```

```
pascal OSErr NewMovieFromHandle
                            (Movie *theMovie, Handle h,
                             short newMovieFlags,
                             Boolean *dataRefWasChanged);
pascal Movie NewMovie       (long newMovieFlags);
pascal OSErr ConvertFileToMovieFile
                            (const FSSpec *inputFile,
                             const FSSpec *outputFile, OSType creator,
                             ScriptCode scriptTag, short *resID,
                             long flags, ComponentInstance userComp,
                             MovieProgressProcPtr proc, long refcon);
pascal OSErr ConvertMovieToFile
                            (Movie theMovie, Track onlyTrack,
                             const FSSpec *outputFile, OSType fileType,
                             OSType creator, ScriptCode scriptTag,
                             short *resID, long flags,
                             ComponentInstance userComp);
pascal void DisposeMovie    (Movie theMovie);
pascal OSErr CreateMovieFile
                            (const FSSpec *fileSpec, OSType creator,
                             ScriptCode scriptTag,
                             long createMovieFileFlags, short *resRefNum,
                             Movie *newMovie);
pascal OSErr OpenMovieFile  (const FSSpec *fileSpec, short *resRefNum,
                             char perms);
pascal OSErr CloseMovieFile
                            (short resRefNum);
pascal OSErr DeleteMovieFile
                            (const FSSpec *fileSpec);
```

Saving Movies

```
pascal Boolean HasMovieChanged
                            (Movie theMovie);
pascal void ClearMovieChanged
                            (Movie theMovie);
pascal OSErr AddMovieResource
                            (Movie theMovie, short resRefNum, short *resId,
                             const StringPtr resName);
pascal OSErr UpdateMovieResource
                            (Movie theMovie, short resRefNum, short resId,
                             const StringPtr resName);
pascal OSErr RemoveMovieResource
                            (short resRefNum, short resId);
```

```
pascal OSErr PutMovieIntoHandle
                            (Movie theMovie, Handle publicMovie);
pascal void FlattenMovie    (Movie theMovie, long movieFlattenFlags,
                             const FSSpec *theFile,
                             OSType creator, ScriptCode scriptTag,
                             long createMovieFileFlags, short *resId,
                             const StringPtr resName);
pascal Movie FlattenMovieData
                            (Movie theMovie, long movieFlattenFlags,
                             const FSSpec *theFile,
                             OSType creator, ScriptCode scriptTag,
                             long createMovieFileFlags);
pascal OSErr NewMovieFromDataFork
                            (Movie *theMovie, short fRefNum,
                             long fileOffset, short newMovieFlags,
                             Boolean *dataRefWasChanged);
pascal OSErr PutMovieIntoDataFork
                            (Movie theMovie, short fRefNum, long offset,
                             long maxSize);
```

Controlling Movie Playback

```
pascal void StartMovie      (Movie theMovie);
pascal void StopMovie       (Movie theMovie);
pascal void GoToBeginningOfMovie
                            (Movie theMovie);
pascal void GoToEndOfMovie  (Movie theMovie);
```

Movie Posters and Movie Previews

```
pascal void SetTrackUsage   (Track theTrack, long usage);
pascal long GetTrackUsage   (Track theTrack);
pascal void ShowMoviePoster
                            (Movie theMovie);
pascal void SetPosterBox    (Movie theMovie, const Rect *boxRect);
pascal void GetPosterBox    (Movie theMovie, Rect *boxRect);
pascal void SetMoviePosterTime
                            (Movie theMovie, TimeValue posterTime);
pascal TimeValue GetMoviePosterTime
                            (Movie theMovie);
pascal void PlayMoviePreview
                            (Movie theMovie,
                             MoviePreviewCallOutProc callOutProc,
                             long refcon);
```

```
pascal void SetMoviePreviewMode
                            (Movie theMovie, Boolean usePreview);
pascal Boolean GetMoviePreviewMode
                            (Movie theMovie);
pascal void SetMoviePreviewTime
                            (Movie theMovie, TimeValue previewTime,
                             TimeValue previewDuration);
pascal void GetMoviePreviewTime
                            (Movie theMovie, TimeValue *previewTime,
                             TimeValue *previewDuration);
```

Movies and Your Event Loop

```
pascal void MoviesTask      (Movie theMovie, long maxMilliSecToUse);
pascal Boolean IsMovieDone  (Movie theMovie);
pascal OSErr UpdateMovie    (Movie theMovie);
pascal Boolean PtInMovie    (Movie theMovie, Point pt);
pascal Boolean PtInTrack    (Track theTrack, Point pt);
pascal ComponentResult GetMovieStatus
                            (Movie theMovie, Track *firstProblemTrack);
pascal ComponentResult GetTrackStatus
                            (Track theTrack);
```

Preferred Movie Settings

```
pascal void SetMoviePreferredRate
                            (Movie theMovie, Fixed rate);
pascal Fixed GetMoviePreferredRate
                            (Movie theMovie);
pascal void SetMoviePreferredVolume
                            (Movie theMovie, short volume);
pascal short GetMoviePreferredVolume
                            (Movie theMovie);
```

Enhancing Movie Playback Performance

```
pascal OSErr PrerollMovie   (Movie theMovie, TimeValue time, Fixed Rate);
pascal void SetMovieActiveSegment
                            (Movie theMovie, TimeValue startTime,
                             TimeValue duration);
pascal void GetMovieActiveSegment
                            (Movie theMovie, TimeValue *startTime,
                             TimeValue *duration);
```

```
pascal void SetMoviePlayHints
                            (Movie theMovie, long flags, long flagsMask);
pascal void SetMediaPlayHints
                            (Media theMedia, long flags, long flagsMask);
pascal OSErr LoadMovieIntoRam
                            (Movie theMovie, TimeValue time,
                             TimeValue duration, long flags);
pascal OSErr LoadTrackIntoRam
                            (Track theTrack, TimeValue time,
                             TimeValue duration, long flags);
pascal OSErr LoadMediaIntoRam
                            (Media theMedia, TimeValue time,
                             TimeValue duration, long flags);
pascal OSErr SetMediaShadowSync
                            (Media theMedia, long frameDiffSampleNum
                             long syncSampleNum);
pascal OSErr GetMediaShadowSync
                            (Media theMedia, long frameDiffSampleNum
                             long *syncSampleNum);
```

Disabling Movies and Tracks

```
pascal void SetMovieActive  (Movie theMovie, Boolean active);
pascal Boolean GetMovieActive
                            (Movie theMovie);
pascal void SetTrackEnabled
                            (Track theTrack, Boolean isEnabled);
pascal Boolean GetTrackEnabled
                            (Track theTrack);
```

Generating Pictures From Movies

```
pascal PicHandle GetMoviePict
                            (Movie theMovie, TimeValue time);
pascal PicHandle GetMoviePosterPict
                            (Movie theMovie);
pascal PicHandle GetTrackPict
                            (Track theTrack, TimeValue time);
```

Creating Tracks and Media Structures

```
pascal Track NewMovieTrack  (Movie theMovie, Fixed width, Fixed height,
                             short trackVolume);
pascal void DisposeMovieTrack
                            (Track theTrack);
```

```
pascal Media NewTrackMedia  (Track theTrack, OSType mediaType,
                             TimeScale timeScale, Handle dataRef,
                             OSType dataRefType);
pascal void DisposeTrackMedia
                            (Media theMedia);
```

Working With Progress and Cover Functions

```
pascal void SetMovieProgressProc
                            (Movie theMovie, MovieProgressProcPtr p, long
                             refcon);
pascal void SetMovieCoverProcs
                            (Movie theMovie, MovieRgnCoverProc uncoverProc,
                             MovieRgnCoverProc coverProc, long refcon);
```

Functions That Modify Movie Properties

Working With Movie Spatial Characteristics

```
pascal void SetMovieGWorld  (Movie theMovie, CGrafPtr port, GDHandle gdh);
pascal void GetMovieGWorld  (Movie theMovie, CGrafPtr *port, GDHandle *gdh);
pascal void SetMovieBox     (Movie theMovie, const Rect *boxRect);
pascal void GetMovieBox     (Movie theMovie, Rect *boxRect);
pascal RgnHandle GetMovieDisplayBoundsRgn
                            (Movie theMovie);
pascal RgnHandle GetMovieSegmentDisplayBoundsRgn
                            (Movie theMovie, TimeValue time,
                             TimeValue duration);
pascal void SetMovieDisplayClipRgn
                            (Movie theMovie, RgnHandle theClip);
pascal RgnHandle GetMovieDisplayClipRgn
                            (Movie theMovie);
pascal RgnHandle GetTrackDisplayBoundsRgn
                            (Track theTrack);
pascal RgnHandle GetTrackSegmentDisplayBoundsRgn
                            (Track theTrack, TimeValue time, TimeValue
                             duration);
pascal void SetTrackLayer   (Track theTrack, short layer);
pascal short GetTrackLayer  (Track theTrack);
pascal void SetMovieMatrix  (Movie theMovie, const MatrixRecord *matrix);
pascal void GetMovieMatrix  (Movie theMovie, MatrixRecord *matrix);
pascal RgnHandle GetMovieBoundsRgn
                            (Movie theMovie);
```

```
pascal RgnHandle GetTrackMovieBoundsRgn
                            (Track theTrack);
pascal void SetMovieClipRgn
                            (Movie theMovie, RgnHandle theClip);
pascal RgnHandle GetMovieClipRgn
                            (Movie theMovie);
pascal void SetTrackMatrix  (Track theTrack, const MatrixRecord *matrix);
pascal void GetTrackMatrix  (Track theTrack, MatrixRecord *matrix);
pascal RgnHandle GetTrackBoundsRgn
                            (Track theTrack);
pascal void SetTrackDimensions
                            (Track theTrack, Fixed width, Fixed height);
pascal void GetTrackDimensions
                            (Track theTrack, Fixed *width, Fixed *height);
pascal void SetTrackClipRgn
                            (Track theTrack, RgnHandle theClip);
pascal RgnHandle GetTrackClipRgn
                            (Track theTrack);
pascal void SetTrackMatte   (Track theTrack, PixMapHandle theMatte);
pascal PixMapHandle GetTrackMatte
                            (Track theTrack);
pascal void DisposeMatte    (PixMapHandle theMatte);
```

Working With Sound Volume

```
pascal void SetMovieVolume  (Movie theMovie, short volume);
pascal short GetMovieVolume
                            (Movie theMovie);
pascal void SetTrackVolume  (Track theTrack, short volume);
pascal short GetTrackVolume
                            (Track theTrack);
```

Working With Movie Time

```
pascal TimeValue GetMovieDuration
                            (Movie theMovie);
pascal void SetMovieTimeValue
                            (Movie theMovie, TimeValue newtime);
pascal void SetMovieTime    (Movie theMovie, const TimeRecord *newtime);
pascal TimeValue GetMovieTime
                            (Movie theMovie, TimeRecord *currentTime);
pascal void SetMovieRate    (Movie theMovie, Fixed rate);
pascal Fixed GetMovieRate   (Movie theMovie);
```

```
pascal void SetMovieTimeScale
                            (Movie theMovie, TimeScale timeScale);
pascal TimeScale GetMovieTimeScale
                            (Movie theMovie);
pascal TimeBase GetMovieTimeBase
                            (Movie theMovie);
```

Working With Track Time

```
pascal TimeValue GetTrackDuration
                            (Track theTrack);
pascal void SetTrackOffset  (Track theTrack, TimeValue movieOffsetTime);
pascal TimeValue GetTrackOffset
                            (Track theTrack);
pascal TimeValue TrackTimeToMediaTime
                            (TimeValue value, Track theTrack);
```

Working With Media Time

```
pascal TimeValue GetMediaDuration
                            (Media theMedia);
pascal void SetMediaTimeScale
                            (Media theMedia, TimeScale timeScale);
pascal TimeScale GetMediaTimeScale
                            (Media theMedia);
```

Finding Interesting Times

```
pascal void GetMovieNextInterestingTime
                            (Movie theMovie, short interestingTimeFlags,
                             short numMediaTypes, const OSType
                             *whichMediaTypes, TimeValue time, Fixed rate,
                             TimeValue *interestingTime,
                             TimeValue *interestingDuration);
pascal void GetTrackNextInterestingTime
                            (Track theTrack, short interestingTimeFlags,
                             TimeValue time, Fixed rate,
                             TimeValue *interestingTime,
                             TimeValue *interestingDuration);
pascal void GetMediaNextInterestingTime
                            (Media theMedia, short interestingTimeFlags,
                             TimeValue time, Fixed rate,
                             TimeValue *interestingTime,
                             TimeValue *interestingDuration);
```

Locating a Movie's Tracks and Media Structures

```
pascal long GetMovieTrackCount
                            (Movie theMovie);
pascal Track GetMovieIndTrack
                            (Movie theMovie, long index);
pascal Track GetMovieTrack  (Movie theMovie, long trackID);
pascal long GetTrackID      (Track theTrack);
pascal Movie GetTrackMovie  (Track theTrack);
pascal Media GetTrackMedia  (Track theTrack);
pascal Track GetMediaTrack  (Media theMedia);
```

Working With Alternate Tracks

```
pascal void SetMovieLanguage(Movie theMovie, long language);
pascal void SelectMovieAlternates
                            (Movie theMovie);
pascal void SetAutoTrackAlternatesEnabled
                            (Movie theMovie, Boolean enable);
pascal void SetTrackAlternate
                            (Track theTrack, Track alternateT);
pascal Track GetTrackAlternate
                            (Track theTrack);
pascal void SetMediaLanguage
                            (Media theMedia, short language);
pascal short GetMediaLanguage
                            (Media theMedia);
pascal void SetMediaQuality
                            (Media theMedia, short quality);
pascal short GetMediaQuality
                            (Media theMedia);
```

Working With Data References

```
pascal OSErr AddMediaDataRef
                            (Media theMedia, short *index, Handle dataRef,
                             OSType dataRefType);
pascal OSErr SetMediaDataRef
                            (Media theMedia, short index, Handle dataRef,
                             OSType dataRefType);
```

```
pascal OSErr GetMediaDataRefCount
                            (Media theMedia, short *count);
pascal OSErr GetMediaDataRef
                            (Media theMedia, short index, Handle *dataRef,
                             OSType *dataRefType, long *dataRefAttributes);
```

Determining Movie Creation and Modification Time

```
pascal unsigned long GetMovieCreationTime
                            (Movie theMovie);
pascal unsigned long GetMovieModificationTime
                            (Movie theMovie);
pascal unsigned long GetTrackCreationTime
                            (Track theTrack);
pascal unsigned long GetTrackModificationTime
                            (Track theTrack);
pascal unsigned long GetMediaCreationTime
                            (Media theMedia);
pascal unsigned long GetMediaModificationTime
                            (Media theMedia);
```

Working With Media Samples

```
pascal long GetMovieDataSize
                            (Movie theMovie, TimeValue startTime,
                             TimeValue duration);
pascal long GetTrackDataSize
                            (Track theTrack, TimeValue startTime,
                             TimeValue duration);
pascal long GetMediaDataSize
                            (Media theMedia, TimeValue startTime,
                             TimeValue duration);
pascal long GetMediaSampleCount
                            (Media theMedia);
pascal long GetMediaSampleDescriptionCount
                            (Media theMedia);
pascal void GetMediaSampleDescription
                            (Media theMedia, long index,
                             SampleDescriptionHandle descH);
pascal OSErr SetMediaSampleDescription
                            (Media theMedia, long index,
                             SampleDescriptionHandle descH);
```

```
pascal void MediaTimeToSampleNum
                            (Media theMedia, TimeValue time,
                             long *sampleNum, TimeValue *sampleTime,
                             TimeValue *sampleDuration);
pascal void SampleNumToMediaTime
                            (Media theMedia, long logicalSampleNum,
                             TimeValue *sampleTime,
                             TimeValue *sampleDuration);
```

Working With Movie User Data

```
pascal UserData GetMovieUserData
                            (Movie theMovie);
pascal UserData GetTrackUserData
                            (Track theTrack);
pascal UserData GetMediaUserData
                            (Media theMedia);
pascal long GetNextUserDataType
                            (UserData theUserData, OSType udType);
pascal short CountUserDataType
                            (UserData theUserData, OSType udType);
pascal OSErr AddUserData    (UserData theUserData, Handle data,
                             OSType udType);
pascal OSErr GetUserData    (UserData theUserData, Handle data,
                             OSType udType,long index);
pascal OSErr RemoveUserData
                            (UserData theUserData, OSType udType,
                             long index);
pascal OSErr AddUserDataText
                            (UserData theUserData, Handle data, OSType
                             udType, long index, short itlRegionTag);
pascal OSErr GetUserDataText
                            (UserData theUserData, Handle data,
                             OSType udType, long index, short itlRegionTag);
pascal OSErr RemoveUserDataText
                            (UserData theUserData, OSType udType,
                             long index, short itlRegionTag);
pascal OSErr SetUserDataItem
                            (UserData theUserData, void *data, long size,
                             OSType udType, long index);
pascal OSErr GetUserDataItem
                            (UserData theUserData, void *data, long size,
                             OSType udType, long index);
```

```
pascal OSErr NewUserData    (UserData *theUserData);
pascal OSErr DisposeUserData
                            (UserData theUserData);
pascal OSErr PutUserDataIntoHandle
                            (UserData theUserData, Handle h);
pascal OSErr NewUserDataFromHandle
                            (Handle h, UserData *theUserData);
```

Functions for Editing Movies

Editing Movies

```
pascal OSErr PutMovieOnScrap
                            (Movie theMovie, long movieScrapFlags);
pascal Movie NewMovieFromScrap
                            (long newMovieFlags);
pascal void SetMovieSelection
                            (Movie theMovie, TimeValue selectionTime,
                             TimeValue selectionDuration);
pascal void GetMovieSelection
                            (Movie theMovie, TimeValue *selectionTime,
                             TimeValue *selectionDuration);
pascal Movie CutMovieSelection
                            (Movie theMovie);
pascal Movie CopyMovieSelection
                            (Movie theMovie);
pascal void PasteMovieSelection
                            (Movie theMovie, Movie src);
pascal void AddMovieSelection
                            (Movie theMovie, Movie src);
pascal void ClearMovieSelection
                            (Movie theMovie);
pascal Component IsScrapMovie
                            (Track targetTrack);
pascal OSErr PasteHandleIntoMovie
                            (Handle h, OSType handleType, Movie theMovie,
                             long flags, ComponentInstance userComp);
pascal OSErr PutMovieIntoTypedHandle
                            (Movie theMovie, Track targetTrack,
                             OSType handleType, Handle publicMovie,
                             TimeValue start, TimeValue dur,
                             long flags, ComponentInstance userComp);
```

Undo for Movies

```
pascal MovieEditState NewMovieEditState
                            (Movie theMovie);
pascal OSErr UseMovieEditState
                            (Movie theMovie, MovieEditState toState);
pascal OSErr DisposeMovieEditState
                            (MovieEditState state);
```

Low-Level Movie-Editing Functions

```
pascal OSErr InsertMovieSegment
                            (Movie srcMovie, Movie dstMovie,
                             TimeValue srcIn,
                             TimeValue srcDuration, TimeValue dstIn);
pascal OSErr InsertEmptyMovieSegment
                            (Movie dstMovie, TimeValue dstIn,
                             TimeValue dstDuration);
pascal OSErr DeleteMovieSegment
                            (Movie theMovie, TimeValue in, TimeValue
                             duration);
pascal OSErr ScaleMovieSegment
                            (Movie theMovie, TimeValue in,
                             TimeValue oldDuration, TimeValue newDuration);
pascal OSErr CopyMovieSettings
                            (Movie srcMovie, Movie dstMovie);
```

Editing Tracks

```
pascal OSErr InsertTrackSegment
                            (Track srcTrack, Track dstTrack,
                             TimeValue srcIn, TimeValue srcDuration,
                             TimeValue dstIn);
pascal OSErr InsertEmptyTrackSegment
                            (Track dstTrack, TimeValue dstIn,
                             TimeValue dstDuration);
pascal OSErr InsertMediaIntoTrack
                            (Track theTrack, TimeValue trackStart,
                             TimeValue mediaTime,
                             TimeValue mediaDuration, Fixed mediaRate);
pascal OSErr DeleteTrackSegment
                            (Track theTrack, TimeValue in,
                             TimeValue duration);
```

```
pascal OSErr ScaleTrackSegment
                            (Track theTrack, TimeValue in,
                             TimeValue oldDuration, TimeValue newDuration);
pascal OSErr CopyTrackSettings
                            (Track srcTrack, Track dstTrack);
pascal Fixed GetTrackEditRate
                            (Track theTrack, TimeValue atTime);
```

Undo for Tracks

```
pascal TrackEditState NewTrackEditState
                            (Track theTrack);
pascal OSErr UseTrackEditState
                            (Track theTrack, TrackEditState state);
pascal OSErr DisposeTrackEditState
                            (TrackEditState state);
```

Adding Samples to Media Structures

```
pascal OSErr BeginMediaEdits
                            (Media theMedia);
pascal OSErr EndMediaEdits  (Media theMedia);
pascal OSErr AddMediaSample (Media theMedia, Handle dataIn, long inOffset,
                             unsigned long size,
                             TimeValue durationPerSample,
                             SampleDescriptionHandle sampleDescriptionH,
                             long numberOfSamples, short sampleFlags,
                             TimeValue *sampleTime);
pascal OSErr AddMediaSampleReference
                            (Media theMedia, long dataOffset,
                             unsigned long size,
                             TimeValue durationPerSample,
                             SampleDescriptionHandle sampleDescriptionH,
                             long numberOfSamples, short sampleFlags,
                             TimeValue *sampleTime);
pascal OSErr GetMediaSample
                            (Media theMedia, Handle dataOut,
                             long maxSizeToGrow, long *size, TimeValue time,
                             TimeValue *sampleTime,
                             TimeValue *durationPerSample,
                             SampleDescriptionHandle sampleDescriptionH,
                             long *sampleDescriptionIndex,
                             long maxNumberOfSamples,
                             long *numberOfSamples, short *sampleFlags);
```

```
pascal OSErr GetMediaSampleReference
                            (Media theMedia, long *dataOffset, long *size,
                             TimeValue time, TimeValue *sampleTime,
                             TimeValue *durationPerSample,
                             SampleDescriptionHandle sampleDescriptionH,
                             long *sampleDescriptionIndex,
                             long maxNumberOfSamples,
                             long *numberOfSamples, short *sampleFlags);
```

Media Functions

Selecting Media Handlers

```
pascal void GetMediaHandlerDescription
                            (Media theMedia, OSType *mediaType,
                             Str255 creatorName,
                             OSType *creatorManufacturer);
pascal MediaHandler GetMediaHandler
                            (Media theMedia);
pascal OSErr SetMediaHandler
                            (Media theMedia, MediaHandlerComponent mH);
pascal void GetMediaDataHandlerDescription
                            (Media theMedia, short index, OSType *dhType,
                             Str255 creatorName,
                             OSType *creatorManufacturer);
pascal DataHandler GetMediaDataHandler
                            (Media theMedia, short index);
pascal OSErr SetMediaDataHandler
                            (Media theMedia, short index,
                             DataHandlerComponent dataHandler);
```

Video Media Handler Functions

```
pascal HandlerError SetVideoMediaGraphicsMode
                            (MediaHandler mh, long graphicsMode,
                             const RGBColor *opColor);
pascal HandlerError GetVideoMediaGraphicsMode
                            (MediaHandler mh, long *graphicsMode,
                             RGBColor *opColor);
```

Sound Media Handler Functions

```
pascal HandlerError SetSoundMediaBalance
                            (MediaHandler mh, short balance);
pascal HandlerError GetSoundMediaBalance
                            (MediaHandler mh, short *balance);
```

Text Media Handler Functions

```
pascal ComponentResult AddTextSample
                            (MediaHandler mh, Ptr text,
                             unsigned long size,  short fontNum,
                             short fontSize,  Style textFace,
                             RGBColor *textColor,
                             RGBColor *backColor,
                             short textJustification,
                             Rect *textBox,  long displayFlags,
                             TimeValue scrollDelay,
                             short hiliteStart, short hiliteEnd,
                             RGBColor rgbHiliteColor,
                             TimeValue duration, TimeValue *sampleTime);
pascal ComponentResult AddTESample
                            (MediaHandler mh, TEHandle hTE,
                             RGBColor *backColor, short textJustification,
                             Rect *textBox, long displayFlags,
                             TimeValue scrollDelay, short hiliteStart,
                             short hiliteEnd, RGBColor rgbHiliteColor,
                             TimeValue duration,
                             TimeValue *sampleTime);
pascal ComponentResult AddHiliteSample
                            (MediaHandler mh, short hiliteStart,
                             short hiliteEnd,
                             RGBColor *rgbHiliteColor, TimeValue duration,
                             TimeValue *sampleTime);
pascal ComponentResult FindNextText
                            (MediaHandler mh, Ptr Text, long size,
                             short findFlags, TimeValue startTime,
                             TimeValue *foundTime, TimeValue *foundDuration,
                             long *offset);
pascal ComponentResult HiliteTextSample
                            (MediaHandler mh, TimeValue sampleTime,
                             short hiliteStart, short hiliteEnd,
                             RGBColor *rgbHiliteColor);
```

```
pascal ComponentResult SetTextProc
                            (MediaHandler mh, TextMediaProcPtr TextProc,
                             long refcon);
```

Functions for Creating File Previews

```
pascal OSErr MakeFilePreview
                            (short resRefNum,
                             ProgressProcRecordPtr progress);
pascal OSErr AddFilePreview
                            (short resRefNum, OSType previewType,
                             Handle previewData);
```

Functions for Displaying File Previews

```
pascal void SFGetFilePreview
                            (Point where, ConstStr255Param prompt,
                             FileFilterProcPtr fileFilter, short numTypes,
                             SFTypeList typeList, DlgHookProcPtr dlgHook,
                             SFReply *reply);
pascal void SFPGetFilePreview
                            (Point where, ConstStr255Param prompt,
                             FileFilterProcPtr fileFilter, short numTypes,
                             SFTypeList typeList, DlgHookProcPtr dlgHook,
                             SFReply *reply, short dlgID,
                             ModalFilterProcPtr filterProc);
pascal void StandardGetFilePreview
                            (FileFilterProcPtr fileFilter, short numTypes,
                             SFTypeList typeList, StandardFileReply *reply);
pascal void CustomGetFilePreview
                            (FileFilterYDProcPtr fileFilter,
                             short numTypes, SFTypeList typeList,
                             StandardFileReply *reply, short dlgID,
                             Point where, DlgHookYDProcPtr dlgHook,
                             ModalFilterYDProcPtr filterProc,
                             short *activeList,
                             ActivateYDProcPtr activateProc,
                             void *yourDataPtr);
```

Time Base Functions

Creating and Disposing of Time Bases

```
pascal TimeBase NewTimeBase
                            (void);
pascal void DisposeTimeBase
                            (TimeBase tb);
pascal void SetMovieMasterClock
                            (Movie theMovie, Component clockMeister,
                             const TimeRecord *slaveZero);
pascal void SetMovieMasterTimeBase
                            (Movie theMovie, TimeBase tb,
                             const TimeRecord *slaveZero);
pascal void SetTimeBaseMasterClock
                            (TimeBase slave, Component clockMeister,
                             const TimeRecord *slaveZero);
pascal ComponentInstance GetTimeBaseMasterClock
                            (TimeBase tb);
pascal void SetTimeBaseMasterTimeBase
                            (TimeBase slave, TimeBase master,
                             const TimeRecord *slaveZero);
pascal TimeBase GetTimeBaseMasterTimeBase
                            (TimeBase tb);
pascal void SetTimeBaseZero
                            (TimeBase tb, TimeRecord *zero);
```

Working With Time Base Values

```
pascal void SetTimeBaseTime
                            (TimeBase tb, const TimeRecord *tr);
pascal void SetTimeBaseValue
                            (TimeBase tb, TimeValue t, TimeScale s);
pascal TimeValue GetTimeBaseTime
                            (TimeBase tb, TimeScale s, TimeRecord *tr);
pascal void SetTimeBaseRate
                            (TimeBase tb, Fixed r);
pascal Fixed GetTimeBaseRate
                            (TimeBase tb);
```

```
pascal Fixed GetTimeBaseEffectiveRate
                            (TimeBase tb);
pascal void SetTimeBaseStartTime
                            (TimeBase tb, const TimeRecord *tr);
pascal TimeValue GetTimeBaseStartTime
                            (TimeBase tb, TimeScale s, TimeRecord *tr);
pascal void SetTimeBaseStopTime
                            (TimeBase tb, const TimeRecord *tr);
pascal TimeValue GetTimeBaseStopTime
                            (TimeBase tb, TimeScale s, TimeRecord *tr);
pascal void SetTimeBaseFlags
                            (TimeBase tb, long timeBaseFlags);
pascal long GetTimeBaseFlags
                            (TimeBase tb);
pascal long GetTimeBaseStatus
                            (TimeBase tb, TimeRecord *unpinnedTime);
```

Working With Times

```
pascal void AddTime         (TimeRecord *dst, const TimeRecord *src);
pascal void SubtractTime    (TimeRecord *dst, const TimeRecord *src);
pascal void ConvertTime     (TimeRecord *inout, TimeBase newBase);
pascal void ConvertTimeScale
                            (TimeRecord *inout, TimeScale newScale);
```

Time Base Callback Functions

```
pascal QTCallBack NewCallBack
                            (TimeBase tb, short cbType);
pascal OSErr CallMeWhen     (QTCallBack cb,
                             QTCallBackProc callBackProc, long refcon,
                             long param1, long param2, long param3);
pascal void CancelCallBack  (QTCallBack cb);
pascal void DisposeCallBack
                            (QTCallBack cb);
pascal TimeBase GetCallBackTimeBase
                            (QTCallBack cb);
pascal short GetCallBackType
                            (QTCallBack cb);
```

Matrix Functions

```
pascal void SetIdentityMatrix
                            (MatrixRecord *matrix);
pascal short GetMatrixType  (MatrixRecordPtr m);
pascal void CopyMatrix      (MatrixRecordPtr m1, MatrixRecord *m2);
pascal Boolean EqualMatrix  (const MatrixRecord *m1,
                             const MatrixRecord *m2);
pascal void TranslateMatrix
                            (MatrixRecord *m, Fixed deltaH, Fixed deltaV);
pascal void ScaleMatrix     (MatrixRecord *m, Fixed scaleX,
                             Fixed scaleY, Fixed aboutX, Fixed aboutY);
pascal void RotateMatrix    (MatrixRecord *m, Fixed degrees,
                             Fixed aboutX, Fixed aboutY);
pascal void SkewMatrix      (MatrixRecord *m, Fixed skewX, Fixed skewY,
                             Fixed aboutX, Fixed aboutY);
pascal void ConcatMatrix    (MatrixRecord *a, MatrixRecord *b);
pascal Boolean InverseMatrix
                            (MatrixRecord *m, MatrixRecord *im);
pascal OSErr TransformPoints
                            (MatrixRecord *mp, Point *pt1, long count);
pascal OSErr TransformFixedPoints
                            (MatrixRecord *m, FixedPoint *fpt, long count);
pascal Boolean TransformRect
                            (MatrixRecord *m, Rect *r, FixedPoint *fpp);
pascal Boolean TransformFixedRect
                            (MatrixRecord *m, FixedRect *fr,
                             FixedPoint *fpp);
pascal OSErr TransformRgn   (MatrixRecordPtr mp, RgnHandle r);
pascal void RectMatrix      (MatrixRecord *matrix, Rect *srcRect,
                             Rect *dstRect);
pascal void MapMatrix       (MatrixRecord *matrix, Rect *fromRect,
                             Rect *toRect);
```

Application-Defined Functions

Progress Functions

```
pascal  OSErr MyProgressProc
                            (Movie theMovie, short message,
                             short whatOperation,
                             Fixed percentDone, long refcon);
```

Cover Functions

```
pascal OSErr MyCoverProc    (Movie theMovie, RgnHandle changedRgn,
                             long refcon);
```

Error-Notification Functions

```
pascal void MyErrProc       (OSErr theErr, long refcon);
```

Movie Callout Functions

```
pascal Boolean MyCallOutProc
                            (long refcon);
```

File Filter Functions

```
pascal Boolean MyFileFilter
                            (ParmBlkPtr parmBlock);
```

Custom Dialog Functions

```
pascal short MyDlgHook      (short item, DialogPtr theDialog,
                             Ptr myDataPtr);
```

Modal-Dialog Filter Functions

```
pascal Boolean MyModalFilter
                            (DialogPtr theDialog, EventRecord *theEvent,
                             short itemHit, Ptr myDataPtr);
```

Standard File Activation Functions

```
pascal void MyActivateProc  (DialogPtr theDialog, short itemNo, Boolean
                             activating, Ptr myDataPtr);
```

Callback Event Functions

```
pascal void MyCallBackProc  (QTCallBack cb, long refcon);
```

Text Functions

```
pascal OSErr MyTextProc     (Handle theText, Movie theMovie,
                             short *displayFlag, long refcon);
```

Pascal Summary

Constants

```
CONST
      kFix1                         = $00010000;{fixed point value equal }
                                                { to 1.0}
      gestaltQuickTime              = 'qtim';   {Movie Toolbox availability}

      MovieFileType                 = 'MooV';   {movie file type}

      VideoMediaType                = 'vide';   {video media type}
      SoundMediaType                = 'soun';   {sound media type}
      MediaHandlerType              = 'mhlr';   {media handler type}
      DataHandlerType               = 'dhlr';   {data handler type}
      TextMediaType                 = 'text';   {text media type}
      GenericMediaType              = 'gnrc';   {base media handler type}

      DoTheRightThing = 0L                      {indicates default flag }
                                                { setting for Movie }
                                                { Toolbox functions}

      {progress procedure messages}
      movieProgressOpen             = 0;        {start of a long operation}
      movieProgressUpdatePercent    = 1;        {completion data to }
                                                { procedure}
      movieProgressClose            = 2;        {end of a long operation}

      {progress procedure operations that indicate which routine }
      { your application has called}
      progressOpFlatten             = 1;        {FlattenMovie or }
                                                { FlattenMovieData}
      progressOpInsertTrackSegment  = 2;        {InsertTrackSegment}
      progressOpInsertMovieSegment  = 3;        {InsertMovieSegment}
      progressOpPaste               = 4;        {PasteMovieSelection}
      progressOpAddMovieSelection   = 5;        {AddMovieSelection}
      progressOpCopy                = 6;        {CopyMovieSelection}
```

```
progressOpCut                 = 7;        {CutMovieSelection}
progressOpLoadMovieIntoRam    = 8;        {LoadMovieIntoRam}
progressOpLoadTrackIntoRam    = 9;        {LoadTrackIntoRam}
progressOpLoadMediaIntoRam    = 10;       {LoadMediaIntoRam}
progressOpImportMovie         = 11;       {ConvertFileToMovieFile}
progressOpExportMovie         = 12;       {ConvertMovieFile}

{NewMovie function flags}
newMovieActive                    = $1;   {is new movie active?}
newMovieDontResolveDataRefs       = $2;   {how data references are }
                                          { resolved in movie resource}
newMovieDontAskUnresolvedDataRefs = $4;   {is user asked to locate }
                                          { files? }
newMovieDontAutoAlternate         = $8;   {are enabled tracks }
                                          { selected from alternate }
                                          { groups?}

{sound volume values in trackVolume parameter of NewMovieTrack }
{ function}
kFullVolume                       = $100; {full, natural volume }
                                          { 8.8 format}
kNoVolume                         = 0;    {sets track to no volume}

{constants for whichMediaTypes parameter of }
{ GetMovieNextInterestingTime function}
VisualMediaCharacteristic 'eyes'          {visual media type}
AudioMediaCharacteristic 'ears'           {audio media type}

{track usage flags in SetTrackUsage procedure}
trackUsageInMovie                 = $2;   {track is used in movie}
trackUsageInPreview               = $4;   {track is used in preview}
trackUsageInPoster                = $8;   {track is used in poster}

{media sample flags in AddMediaSample function}
mediaSampleNotSync                = 1;    {sample to be added not a }
                                          { sync sample}
mediaSampleShadowSync             = 2;    {sample is a shadow }
                                          { sync sample}

{media quality settings in quality parameter of }
{ SetMediaQuality procedure}
mediaQualityDraft                 = $0000; {lowest quality level}
```

```
mediaQualityNormal                = $0040; {acceptable quality level}
mediaQualityBetter                = $0080; {better quality level}
mediaQualityBest                  = $00C0; {best quality level}

{interesting times flags in interestingTimeFlags parameter }
{ of GetMovieNextInterestingTime procedure specify searching criteria}
nextTimeMediaSample            = $1;       {next sample in movie's }
                                           { media}
nextTimeMediaEdit              = $2;       {next sample group in media}
nextTimeTrackEdit              = $4;       {sample for next entry }
                                           { in edit list}
nextTimeSyncSample             = $8;       {next sync sample in }
                                           { movie's media}
nextTimeEdgeOK                 = $2000;    {get specified time }
                                           { element data}
nextTimeIgnoreActiveSegment    = $4000;    {outside active segment}

{flag for resID parameter of NewMovieFile function}
movieInDataForkResID           = -1;       {magic resource ID}

{movie-creation flags from CreateMovieFile function}
createMovieFileDeleteCurFile   = $80000000;{delete existing file?}
createMovieFileDontCreateMovie = $40000000;{new movie created?}
createMovieFileDontOpenFile    = $20000000;{new movie file opened?}

{movie-flattening flags from FlattenFlags procedure}
flattenAddMovieToDataFork      = $1;       {movie in data fork of }
                                           { new movie file}
flattenActiveTracksOnly        = $4;       {enabled tracks added }
                                           { to movie file}
flattenDontInterleaveFlatten   = $8;       {disables data }
                                           { storage optimizations}

{movie scrap flags from PutMovieOnScrap function}
movieScrapDontZeroScrap        = $1;   {is scrap cleared before }
                                       { movie on scrap?}
movieScrapOnlyPutMovie         = $2;   {are other items on }
                                       { scrap with movie?}

mediaHandlerFlagGenericClient = 1;  {component flag--should be set }
                                    { for all media handlers }
                                    { components that use generic }
                                    { media handlers}
```

```
{callback flags from CallMeWhen function specify when callback }
{ should be called}
triggerTimeFwd                = $0001;    {time is at positive rate}
triggerTimeBwd                = $0002;    {time is at negative rate}
triggerTimeEither             = $0003;    {without regard to rate}
triggerRateLT                 = $0004;    {whenever rate changes}
triggerRateGT                 = $0008;    {rate change less than }
                                          { param2}
triggerRateEqual              = $0010;    {rate change equal to }
                                          { param2}
triggerRateLTE                = $0014;    {rate change less than or }
                                          { equal to param2}
triggerRateGTE            = $0018; {rate change greater than }
                                   { or equal to param2 specification}
triggerRateNotEqual       = $001C; {rate change not equal to param2}
triggerRateChange         = 0;     {whenever rate changes}
triggerAtStart            = $0001; {at start time}
triggerAtStop             = $0002; {at stop time}

{flags returned by GetTimeBaseStatus function specify where }
{ time value in time record lies}
timeBaseBeforeStartTime   = 1;     {before start time of time base}
timeBaseAfterStopTime     = 2;     {after stop time of time base}

{values for cbType parameter of NewCallBack function specify when }
{ event can be invoked}
callBackAtTime            = 1;     {at a specified time}
callBackAtRate            = 2;     {rate for time base reaches value}
callBackAtTimeJump        = 3;     {when time value changes }
                                   { by amount differing from }
                                   { its rate}
callBackAtExtremes        = 4;     {at start time, at stop time, }
                                   { or both}
callBackAtInterrupt       = $8000; {at interrupt time}

{values for callBackFlags field of QuickTime callback header record }
{ used by clock components to communicate scheduling information }
{ about a callback event to the Movie Toolbox}
qtcbNeedsRateChanges      = 1;     {rate changes}
qtcbNeedsTimeChanges      = 2;     {time changes}
qtcbNeedsStartStopChanges = 4;;    {changes in time base start/stop}
```

```
{dialog items to include in dialog box definition for use with }
{ SFPGetFilePreview function}
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}

{control flags for timeBaseFlags parameter of SetTimeBaseFlags }
{ function}
loopTimeBase               = 1;     {whether time base loops}
palindromeLoopTimeBase     = 2;     {whether time base loops }
                                    { in palindrome fashion}

{flags for LoadIntoRAM functions}
keepInRam                  = 1;     {load and make so data cannot be }
                                    { purged}
unkeepInRam                = 2;     {mark data so it can be purged}
flushFromRam               = 4;     {empty handles and purge data }
                                    { from memory}
loadForwardTrackEdits      = 8;     {load only data around track }
                                    { edits--play movie forward}
loadBackwardTrackEdits     = 16;    {load only data around track}
                                    { edits--play movie in reverse}

{flag for PasteHandleIntoMovie function}
pasteInParallel            = 1;     {changes function to take }
                                    { contents and type of handle }
                                    { and add to movie}

{text description display flags used in AddTextSample and }
{ AddTESample functions}
dfDontDisplay              = 1;     {don't display the text}
dfDontAutoScale            = 2;     {don't scale text as track }
                                    { boundaries grow or shrink}
dfClipToTextBox            = 4;     {clip update to the text box}
dfUseMovieBGColor          = 8;     {set text background to }
                                    { movie's background color}
dfShrinkTextBoxToFit       = 16;    {compute minimum box to fit }
                                    { the sample}
dfScrollIn                 = 32;    {scroll text in until last }
                                    { of text is in view}
dfScrollOut                = 64;    {scroll text out until last }
                                    { of text is gone}
```

```
dfHorizScroll              = 128;   {scroll text horizontally}
dfReverseScroll            = 256;   {vertical text scrolls down, }
                                    { horizontal text scrolls }
                                    { backward;justification dependent}

{values returned by the GetMatrixType function}
identityMatrixType         = $00;   {matrix is identity}
translateMatrixType        = $01;   {matrix translates}
scaleMatrixType            = $02;   {matrix scales}
scaleTranslateMatrixType   = $03;   {matrix scales and translates}
linearMatrixType           = $04;   {matrix is general 2 x 2}
linearTranslateMatrixType  = $05;   {matrix is general 2 x 2 }
                                    { and translates}
perspectiveMatrixType      = $06;   {matrix is general 3 x 3}

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

{find flags for FindNextTextFunction}
findTextEdgeOK             = 1;     {OK to find text at specified }
                                    { sample time}
findTextCaseSensitive      = 2;     {case-sensitive search}
findTextReverseSearch      = 4;     {search from sample time backward}
findTextWrapAround         = 8;     {wrap search when beginning or }
                                    { end of movie is reached}

{hints constants for play hints functions}
hintsScrubMode             = $1;    {toolbox can display key frames }
                                    { when movie is repositioned}
hintsAllowInterlace        = $40;   {use interlace option for }
                                    { compressor components}
hintsUseSoundInterp        = $80;   {turns on sound interpolation}
```

Data Types

```
TYPE  Movie          = ^MovieRecord;         {movie identifier}
      Track          = ^TrackRecord;         {track identifer}
      Media          = ^MediaRecord;         {media identifier}
      UserData       = ^UserDataRecord;      {user data list identifier}
      TrackEditState = ^TrackEditStateRecord;{track edit state identifier}
      MovieEditState = ^MovieEditStateRecord;{movie edit state identifier}
      TimeValue      = LongInt;              {time value}
```

```
TimeScale      = LongInt;              {time scale in time record}
TimeBase       = ^TimeBaseRecord;      {time base identifier}
TimeBaseStatus = LongInt;              {time base statistics}
QTCallBack     = ^CallBackRecord;      {callback identifier}
QTCallBackProc             = procPtr;        {callback function }
                                             { identifier}
Int64                      = CompTimeValue;  {time value in time }
                                             { record}
playHintsEnum              = LongInt;        {play hints enumeration}
movieFlattenFlagsEnum      = LontInt;        {movie flatteners flags }
                                             { enumeration}
createMovieFileFlagsEnum   = LongInt;        {movie creation flags}
nextTimeFlagsEnum          = Byte;           {next time flags}

Int64 =
RECORD
   hi:         LongInt; {high-order bits of value field in time record}
   lo:         LongInt; {low-order bits of value field in time record}
END;

TimeRecord =
RECORD
   value:         CompTimeValue; {time value as duration or }
                                 { absolute time}
   scale:         TimeScale;     {time scale as time units}
   base:          TimeBase;      {reference to the time base}
END;

SampleDescriptionPtr= ^SampleDescription;{ptr to sample description}
SampleDescriptionHandle = ^SampleDescriptionPtr;{handle to sample }
                                                { description record}

SampleDescription =
RECORD
   descSize:      LongInt;       {total size in bytes of this record}
   dataFormat:    LongInt;       {format of the sample data}
   resvd1:        LongInt;       {reserved--set to 0}
   resvd2:        Integer;       {reserved--set to 0}
   dataRefIndex:  Integer;       {reserved--set to 1}

END;

SoundDescriptionPtr     = ^SoundDescription; {ptr to sound description}
SoundDescriptionHandle  = ^SoundDescriptionPtr; {handle to sound }
                                                { description record}
```

```
      SoundDescription =
         RECORD
            descSize:         LongInt; {total size in bytes of this record}
            dataFormat:       LongInt; {format of the sound data}
            resvd1:           LongInt; {reserved--set to 0}
            resvd2:           Integer; {reserved--set to 0}
            dataRefIndex:     Integer; {reserved--set to 1}
            version:          Integer; {which version is this data? }
                                       { (set to 0)}
            revlevel:         Integer; {which version of the compressor }
                                       { component did this? (set to 0)}
            vendor:           LongInt; {whose compressor component }
                                       { compressed this data? (set to 0)}
            numChannels:      Integer; {number of sound channels}
            sampleSize:       Integer; {number of bits in each sample;}
            compressionID:    Integer; {sound compression used--0 if none}
            packetSize:       Integer; {packet size for compression--0 if }
                                       { no compression}
            sampleRate:       Fixed;   {rate at which sound samples }
                                       { were obtained}
         END;

TextDescriptionPtr = ^TextDescription;
TextDescriptionHandle = ^TextDescriptionPtr;
   TextDescription =
      RECORD
         descSize:            LongInt;       {total size of this text }
                                             { description record}
         dataFormat:          LongInt;       {type of data in this record }
                                             { ('text')}
         resvd1:              LongInt;       {reserved for use by Apple-- }
                                             { set to 0}
         resvd2:              Integer;       {reserved for use by Apple-- }
                                             { set to 0}
         dataRefIndex:        Integer;       {index to data references}
         displayFlags:        LongInt;       {display flags for text}
         textJustification:   LongInt;       {text justification flags}
         bgColor:             RGBColor;      {background color}
         defaultTextBox:      Rect;          {location of the text within }
                                             { track bounds}
         defaultStyle:        ScrpSTElement; {default style (TextEdit }
                                             { record)}
      END;
```

```
MovieProgressProcPtr    = ProcPtr;  {pointer to application-defined }
                                    { movie progress procedure}

MovieRgnCoverProc       = ProcPtr;  {a pointer to application-defined }
                                    { cover procedure}

MediaInformationHandle  = Handle;           {data returned }
                                            { by media handler}
MediaHandler            = ComponentInstance; {media handler}
MediaHandlerComponent   = Component;        {media handler component}
DataHandler             = ComponentInstance; {data handler}
DataHandlerComponent    = Component;        {data handler component}
HandlerError            = ComponentResult;  {error handler}

MovieController         = ComponentInstance; {movie controller}

ErrorProcPtr            = ProcPtr;  {pointer to application-defined }
                                    { error-notification procedure}
MoviePreviewCallOutProc = ProcPtr;  {pointer to application-defined }
                                    { movie preview callout procedure}
TimeBaseFlags           = Char;     {control flags for time base}
QTCallBackProc          = ProcPtr;  {pointer to application-defined }
                                    { callback routine}

QTCallBackHeader =
RECORD
   callBackFlags:       LongInt;    {flags used by clock component to }
                                    { communicate scheduling data }
                                    { about callback to Movie Toolbox}
   reserved1:           LongInt;    {reserved for use by Apple}
   qtPrivate:           PACKED ARRAY[0..39] of Byte;
                                    {reserved for use by Apple}
END;

MatrixRecordPtr         = ^MatrixRecord;  {pointer to matrix record}

MatrixRecord =
RECORD
   matrix:              ARRAY[0..2,0..2] of Fixed;
END;

FixedPoint =
RECORD
```

```
      x:             Fixed;   {point's x coordinate as fixed-point number}
      y:             Fixed;   {point's y coordinate as fixed-point number}
   END;

   FixedRect =
   RECORD
      left:    Fixed;   {x coordinate of rectangle's upper-left corner}
      top:     Fixed;   {y coordinate of rectangle's upper-left corner}
      right:   Fixed;   {x coordinate of rectangle's lower-right corner}
      bottom:  Fixed;   {y coordinate of rectangle's lower-right corner}
   END;
```

Routines for Getting and Playing Movies

Initializing the Movie Toolbox

```
FUNCTION EnterMovies:       OSErr;
PROCEDURE ExitMovies;
```

Error Routines

```
FUNCTION GetMoviesError:    OSErr;
FUNCTION GetMoviesStickyError:
                            OSErr;
PROCEDURE ClearMoviesStickyError;
PROCEDURE SetMoviesErrorProc
                            (errProc: ErrorProcPtr; refcon: LongInt);
```

Movie Routines

```
FUNCTION NewMovieFromFile   (VAR theMovie: Movie; resRefNum: Integer;
                             VAR resId: Integer; resName: Str255;
                             newMovieFlags: Integer;
                             VAR dataRefWasChanged: Boolean): OSErr;
FUNCTION NewMovieFromHandle
                            (VARh: Handle; newMovieFlags: LongInt;
                             VAR dataRefWasChanged: Boolean): OSErr;
FUNCTION NewMovie           (newMovieFlags: LongInt): Movie;
FUNCTION ConvertFileToMovieFile
                            (inputFile: FSSpec; outputFile: FSSpec;
                             creator: OSType; scriptTag: ScriptCode;
                             VAR resID: Integer; flags: LongInt;
                             userComp: ComponentInstance;
                             proc: ProcPtr; refcon: LongInt): OSErr;
```

```
FUNCTION ConvertMovieToFile
                            (theMovie: Movie; onlyTrack: Track;
                             outputFile: FSSpec; fileType: OSType;
                             creator: OSType; scriptTag: ScriptCode;
                             VAR resID: Integer; flags: LongInt;
                             userComp: ComponentInstance): OSErr;
PROCEDURE DisposeMovie      (theMovie: Movie);
FUNCTION CreateMovieFile    (fileSpec: FSSpec; creator: OSType;
                             scriptTag: ScriptCode;
                             createMovieFileFlags: LongInt;
                             VAR resRefNum: Integer;
                             VAR newMovie: Movie): OSErr;
FUNCTION OpenMovieFile      (fileSpec: FSSpec; VAR resRefNum: Integer;
                             perms: SignedByte): OSErr;
FUNCTION CloseMovieFile     (resRefNum: Integer): OSErr;
FUNCTION DeleteMovieFile    (fileSpec: FSSpec): OSErr;
```

Saving Movies

```
FUNCTION HasMovieChanged    (theMovie: Movie): Boolean;
PROCEDURE ClearMovieChanged
                            (theMovie: Movie);
FUNCTION AddMovieResource   (theMovie: Movie; resRefNum: Integer;
                             VAR resId: Integer; resName: Str255): OSErr;
FUNCTION UpdateMovieResource
                            (theMovie: Movie; resRefNum: Integer;
                             VAR resId: Integer; resName: Str255): OSErr;
FUNCTION RemoveMovieResource
                            (resRefNum: Integer; resId: Integer): OSErr;
FUNCTION PutMovieIntoHandle
                            (theMovie: Movie; publicMovie: Handle): OSErr;
PROCEDURE FlattenMovie      (theMovie: Movie; movieFlattenFlags: LongInt;
                             theFile: FSSpec; creator: OSType;
                             scriptTag: ScriptCode;
                             createMovieFileFlags: LongInt;
                             VAR resId: Integer; resName: Str255);
FUNCTION FlattenMovieData   (theMovie: Movie; movieFlattenFlags: LongInt;
                             theFile: FSSpec; creator: OSType;
                             scriptTag: ScriptCode;
                             createMovieFileFlags: LongInt): Movie;
```

```
FUNCTION NewMovieFromDataFork
                            (VAR theMovie: Movie; fRefNum: Integer;
                             fileOffset: Integer; newMovieFlags: Integer;
                             VAR dataRefWasChanged: Boolean): OSErr;
FUNCTION PutMovieIntoDataFork
                            (theMovie: Movie; fRefNum: Integer;
                             offset: LongInt; maxSize: LongInt): OSErr;
```

Controlling Movie Playback

```
PROCEDURE StartMovie        (theMovie: Movie);
PROCEDURE StopMovie         (theMovie: Movie);
PROCEDURE GoToBeginningOfMovie
                            (theMovie: Movie);
PROCEDURE GoToEndOfMovie    (theMovie: Movie);
```

Movie Posters and Movie Previews

```
PROCEDURE SetTrackUsage     (theTrack: Track; usage: LongInt);
FUNCTION GetTrackUsage      (theTrack: Track): LongInt;
PROCEDURE ShowMoviePoster   (theMovie: Movie);
PROCEDURE SetPosterBox      (theMovie: Movie; boxRect: Rect);
PROCEDURE GetPosterBox      (theMovie: Movie; VAR boxRect: Rect);
PROCEDURE SetMoviePosterTime
                            (theMovie: Movie; posterTime: TimeValue);
FUNCTION GetMoviePosterTime
                            (theMovie: Movie): TimeValue;
PROCEDURE PlayMoviePreview  (theMovie: Movie;
                             callOutProc: MoviePreviewCallOutProc;
                             refcon: LongInt);
PROCEDURE SetMoviePreviewMode
                            (theMovie: Movie; usePreview: Boolean);
FUNCTION GetMoviePreviewMode
                            (theMovie: Movie): Boolean;
PROCEDURE SetMoviePreviewTime
                            (theMovie: Movie; previewTime: TimeValue;
                             previewDuration: TimeValue);
PROCEDURE GetMoviePreviewTime
                            (theMovie: Movie; VAR previewTime: TimeValue;
                             VAR previewDuration: TimeValue);
```

Movies and Your Event Loop

```
PROCEDURE MoviesTask        (theMovie: Movie; maxMilliSecToUse: LongInt);
FUNCTION IsMovieDone        (theMovie: Movie): Boolean;
FUNCTION UpdateMovie        (theMovie: Movie): OSErr;
FUNCTION PtInMovie          (theMovie: Movie; pt: Point): Boolean;
FUNCTION PtInTrack          (theTrack: Track; pt: Point): Boolean;
FUNCTION GetMovieStatus     (theMovie: Movie;
                             VAR firstProblemTrack: Track): ComponentResult;
FUNCTION GetTrackStatus     (theTrack: Track): ComponentResult;
```

Preferred Movie Settings

```
PROCEDURE SetMoviePreferredRate
                            (theMovie: Movie; rate: Fixed);
FUNCTION GetMoviePreferredRate
                            (theMovie: Movie): Fixed;
PROCEDURE SetMoviePreferredVolume
                            (theMovie: Movie; volume: Integer);
FUNCTION GetMoviePreferredVolume
                            (theMovie: Movie): Integer;
```

Enhancing Movie Playback Performance

```
FUNCTION PrerollMovie       (theMovie: Movie; time: TimeValue;
                             Rate: Fixed): OSErr;
PROCEDURE SetMovieActiveSegment
                            (theMovie: Movie; startTime: TimeValue;
                             duration: TimeValue);
PROCEDURE GetMovieActiveSegment
                            (theMovie: Movie; VAR startTime: TimeValue;
                             VAR duration: TimeValue);
PROCEDURE SetMoviePlayHints
                            (theMovie: Movie; flags: LongInt;
                             flagsMask: LongInt);
PROCEDURE SetMediaPlayHints
                            (theMedia: Media; flags: LongInt;
                             flagsMask: LongInt);
FUNCTION LoadMovieIntoRam   (theMovie: Movie; time: TimeValue;
                             duration: TimeValue; flags: LongInt): OSErr;
FUNCTION LoadTrackIntoRam   (theTrack: Track; time: TimeValue;
                             duration: TimeValue; flags: LongInt): OSErr;
FUNCTION LoadMediaIntoRam   (theMedia: Media; time: TimeValue;
                             duration: TimeValue; flags: LongInt): OSErr;
```

```
FUNCTION SetMediaShadowSync
                            (theMedia: Media; frameDiffSampleNum: LongInt;
                             syncSampleNum: LongInt): OSErr;
FUNCTION GetMediaShadowSync
                            (theMedia: Media; frameDiffSampleNum: LongInt;
                             VAR syncSampleNum: LongInt): OSErr;
```

Disabling Movies and Tracks

```
PROCEDURE SetMovieActive    (theMovie: Movie; active: Boolean);
FUNCTION GetMovieActive     (theMovie: Movie): Boolean;
PROCEDURE SetTrackEnabled   (theTrack: Track; isEnabled: Boolean);
FUNCTION GetTrackEnabled    (theTrack: Track): Boolean;
```

Generating Pictures From Movies

```
FUNCTION GetMoviePict       (theMovie: Movie; time: TimeValue): PicHandle;
FUNCTION GetMoviePosterPict
                            (theMovie: Movie): PicHandle;
FUNCTION GetTrackPict       (theTrack: Track; time: TimeValue): PicHandle;
```

Creating Tracks and Media Structures

```
FUNCTION NewMovieTrack      (theMovie: Movie; width: Fixed; height: Fixed;
                             trackVolume: Integer): Track;
PROCEDURE DisposeMovieTrack
                            (theTrack: Track);
FUNCTION NewTrackMedia      (theTrack: Track; mediaType: OSType;
                             timeScale: TimeScale; dataRef: Handle;
                             dataRefType: OSType): Media;
PROCEDURE DisposeTrackMedia
                            (theMedia: Media);
```

Working With Progress and Cover Procedures

```
PROCEDURE SetMovieProgressProc
                            (theMovie: Movie; p: MovieProgressProcPtr;
                             refcon: LongInt);
PROCEDURE SetMovieCoverProcs
                            (theMovie: Movie;
                             uncoverProc: MovieRgnCoverProc;
                             coverProc: MovieRgnCoverProc; refcon: LongInt);
```

Routines That Modify Movie Properties

Working With Movie Spatial Characteristics

```
PROCEDURE SetMovieGWorld    (theMovie: Movie; port: CGrafPtr;
                             gdh: GDHandle);
PROCEDURE GetMovieGWorld    (theMovie: Movie; VAR port: CGrafPtr;
                             VAR gdh: GDHandle);
PROCEDURE SetMovieBox       (theMovie: Movie; boxRect: Rect);
PROCEDURE GetMovieBox       (theMovie: Movie; VAR boxRect: Rect);
FUNCTION GetMovieDisplayBoundsRgn
                            (theMovie: Movie): RgnHandle;
FUNCTION GetMovieSegmentDisplayBoundsRgn
                            (theMovie: Movie; time: TimeValue;
                             duration: TimeValue): RgnHandle;
PROCEDURE SetMovieDisplayClipRgn
                            (theMovie: Movie; theClip: RgnHandle);
FUNCTION GetMovieDisplayClipRgn
                            (theMovie: Movie): RgnHandle;
FUNCTION GetTrackDisplayBoundsRgn
                            (theTrack: Track): RgnHandle;
FUNCTION GetTrackSegmentDisplayBoundsRgn
                            (theTrack: Track; time: TimeValue;
                             duration: TimeValue): RgnHandle;
PROCEDURE SetTrackLayer     (theTrack: Track; layer: Integer);
FUNCTION GetTrackLayer      (theTrack: Track): Integer;
PROCEDURE SetMovieMatrix    (theMovie: Movie; matrix: MatrixRecord);
PROCEDURE GetMovieMatrix    (theMovie: Movie; VAR matrix: MatrixRecord);
FUNCTION GetMovieBoundsRgn  (theMovie: Movie): RgnHandle;
FUNCTION GetTrackMovieBoundsRgn
                            (theTrack: Track): RgnHandle;
PROCEDURE SetMovieClipRgn   (theMovie: Movie; theClip: RgnHandle);
FUNCTION GetMovieClipRgn    (theMovie: Movie): RgnHandle;
PROCEDURE SetTrackMatrix    (theTrack: Track; matrix: MatrixRecord);
PROCEDURE GetTrackMatrix    (theTrack: Track; VAR matrix: MatrixRecord);
FUNCTION GetTrackBoundsRgn  (theTrack: Track): RgnHandle;
PROCEDURE SetTrackDimensions
                            (theTrack: Track; width: Fixed; height: Fixed);
PROCEDURE GetTrackDimensions
                            (theTrack: Track; VAR width: Fixed;
                             VAR height: Fixed);
```

```
PROCEDURE SetTrackClipRgn   (theTrack: Track; theClip: RgnHandle);
FUNCTION GetTrackClipRgn    (theTrack: Track): RgnHandle;
PROCEDURE SetTrackMatte     (theTrack: Track; theMatte: PixMapHandle);
FUNCTION GetTrackMatte      (theTrack: Track): PixMapHandle;
PROCEDURE DisposeMatte      (theMatte: PixMapHandle);
```

Working With Sound Volume

```
PROCEDURE SetMovieVolume    (theMovie: Movie; volume: Integer);
FUNCTION GetMovieVolume     (theMovie: Movie): Integer;
PROCEDURE SetTrackVolume    (theTrack: Track; volume: Integer);
FUNCTION GetTrackVolume     (theTrack: Track): Integer;
```

Working With Movie Time

```
FUNCTION GetMovieDuration   (theMovie: Movie): TimeValue;
PROCEDURE SetMovieTimeValue
                            (theMovie: Movie; newtime: TimeValue);
PROCEDURE SetMovieTime      (theMovie: Movie; newtime: TimeRecord);
FUNCTION GetMovieTime       (theMovie: Movie; VAR currentTime: TimeRecord):
                             TimeValue;
PROCEDURE SetMovieRate      (theMovie: Movie; rate: Fixed);
FUNCTION GetMovieRate       (theMovie: Movie): Fixed;
PROCEDURE SetMovieTimeScale
                            (theMovie: Movie; timeScale: TimeScale);
FUNCTION GetMovieTimeScale  (theMovie: Movie): TimeScale;
FUNCTION GetMovieTimeBase   (theMovie: Movie): TimeScale;
```

Working With Track Time

```
FUNCTION GetTrackDuration   (theTrack: Track): TimeValue;
PROCEDURE SetTrackOffset    (theTrack: Track; movieOffsetTime: TimeValue);
FUNCTION GetTrackOffset     (theTrack: Track): TimeValue;
FUNCTION TrackTimeToMediaTime
                            (value: TimeValue; theTrack: Track): TimeValue;
```

Working With Media Time

```
FUNCTION GetMediaDuration   (theMedia: Media): TimeValue;
PROCEDURE SetMediaTimeScale
                            (theMedia: Media; timeScale: TimeScale);
FUNCTION GetMediaTimeScale  (theMedia: Media): TimeScale;
```

Finding Interesting Times

```
PROCEDURE GetMovieNextInterestingTime
                            (theMovie: Movie; interestingTimeFlags: Integer;
                             numMediaTypes: Integer;
                             whichMediaTypes: OSTypePtr; time: TimeValue;
                             rate: Fixed; VAR interestingTime: TimeValue;
                             VAR interestingDuration: TimeValue);
PROCEDURE GetTrackNextInterestingTime
                            (theTrack: Track; interestingTimeFlags: Integer;
                             time: TimeValue; rate: Fixed;
                             VAR interestingTime: TimeValue;
                             VAR interestingDuration: TimeValue);
PROCEDURE GetMediaNextInterestingTime
                            (theMedia: Media; interestingTimeFlags: Integer;
                             time: TimeValue; rate: Fixed;
                             VAR interestingTime: TimeValue;
                             VAR interestingDuration: TimeValue);
```

Locating a Movie's Tracks and Media Structures

```
FUNCTION GetMovieTrackCount
                            (theMovie: Movie): LongInt;
FUNCTION GetMovieIndTrack   (theMovie: Movie; index: LongInt): Track;
FUNCTION GetMovieTrack      (theMovie: Movie; trackID: LongInt): Track;
FUNCTION GetTrackID         (theTrack: Track): LongInt;
FUNCTION GetTrackMovie      (theTrack: Track): Movie;
FUNCTION GetTrackMedia      (theTrack: Track): Media;
FUNCTION GetMediaTrack      (theMedia: Media): Track;
```

Working With Alternate Tracks

```
PROCEDURE SetMovieLanguage  (theMovie: Movie; language: LongInt);
PROCEDURE SelectMovieAlternates
                            (theMovie: Movie);
PROCEDURE SetAutoTrackAlternatesEnabled
                            (theMovie: Movie; enable: Boolean);
PROCEDURE SetTrackAlternate
                            (theTrack: Track; alternateT: Track);
FUNCTION GetTrackAlternate  (theTrack: Track): Track;
PROCEDURE SetMediaLanguage  (theMedia: Media; language: Integer);
FUNCTION GetMediaLanguage   (theMedia: Media): Integer;
PROCEDURE SetMediaQuality   (theMedia: Media; quality: Integer);
FUNCTION GetMediaQuality    (theMedia: Media): Integer;
```

Working With Data References

```
FUNCTION AddMediaDataRef    (theMedia: Media; VAR index: Integer;
                             dataRef: Handle; dataRefType: OSType): OSErr;
FUNCTION SetMediaDataRef    (theMedia: Media; index: Integer;
                             dataRef: Handle; dataRefType: OSType): OSType;
FUNCTION GetMediaDataRefCount
                            (theMedia: Media; VAR count: Integer): OSErr;
FUNCTION GetMediaDataRef    (theMedia: Media; index: Integer;
                             VAR dataRef: Handle; VAR dataRefType: OSType;
                             VAR dataRefAttributes: LongInt): OSErr;
```

Determining Movie Creation and Modification Time

```
FUNCTION GetMovieCreationTime
                            (theMovie: Movie): LongInt;
FUNCTION GetMovieModificationTime
                            (theMovie: Movie): LongInt;
FUNCTION GetTrackCreationTime
                            (theTrack: Track): LongInt;
FUNCTION GetTrackModificationTime
                            (theTrack: Track): LongInt;
FUNCTION GetMediaCreationTime
                            (theMedia: Media): LongInt;
FUNCTION GetMediaModificationTime
                            (theMedia: Media): LongInt;
```

Working With Media Samples

```
FUNCTION GetMovieDataSize   (theMovie: Movie; startTime: TimeValue;
                             duration: TimeValue): LongInt;
FUNCTION GetTrackDataSize   (theTrack: Track; startTime: TimeValue;
                             duration: TimeValue): LongInt;
FUNCTION GetMediaDataSize   (theMedia: Media; startTime: TimeValue;
                             duration: TimeValue): LongInt;
FUNCTION GetMediaSampleCount
                            (theMedia: Media): LongInt;
FUNCTION GetMediaSampleDescriptionCount
                            (theMedia: Media): LongInt;
PROCEDURE GetMediaSampleDescription
                            (theMedia: Media; index: LongInt;
                             descH: SampleDescriptionHandle);
FUNCTION SetMediaSampleDescription
                            (theMedia: Media; index: LongInt;
                             descH: SampleDescriptionHandle): OSErr;
```

```
PROCEDURE MediaTimeToSampleNum
                            (theMedia: Media; time: TimeValue;
                             VAR sampleNum: LongInt;
                             VAR sampleTime: TimeValue;
                             VAR sampleDuration: TimeValue);
PROCEDURE SampleNumToMediaTime
                            (theMedia: Media; logicalSampleNum: LongInt;
                             VAR sampleTime: TimeValue;
                             VAR sampleDuration: TimeValue);
```

Working With Movie User Data

```
FUNCTION GetMovieUserData   (theMovie: Movie): UserData;
FUNCTION GetTrackUserData   (theTrack: Track): UserData;
FUNCTION GetMediaUserData   (theMedia: Media): UserData;
FUNCTION GetNextUserDataType
                            (theUserData: UserData;
                             udType: OSType): LongInt;
FUNCTION CountUserDataType  (theUserData: UserData;
                             udType: OSType): Integer;
FUNCTION AddUserData        (theUserData: UserData; data: Handle;
                             udType: OSType): OSErr;
FUNCTION GetUserData        (theUserData: UserData; data: Handle;
                             udType: OSType; index: LongInt): OSErr;
FUNCTION RemoveUserData     (theUserData: UserData; udType: OSType;
                             index: LongInt): OSErr;
FUNCTION AddUserDataText    (theUserData: UserData; data: Handle;
                             udType: OSType;
                             index: LongInt; itlRegionTag: Integer): OSErr;
FUNCTION GetUserDataText    (theUserData: UserData; data: Handle;
                             udType: OSType; index: LongInt;
                             itlRegionTag: Integer): OSErr;
FUNCTION RemoveUserDataText
                            (theUserData: UserData; udType: OSType;
                             index: LongInt; itlRegionTag: Integer): OSErr;
FUNCTION SetUserDataItem    (theUserData: UserData; data: Ptr;
                             size: LongInt; udType: OSType;
                             index: LongInt): OSErr;
FUNCTION GetUserDataItem    (theUserData: UserData; data: Ptr;
                             size: LongInt; udType: OSType; index: long):
                             OSErr;
FUNCTION NewUserData        (VAR theUserData: UserData): OSErr;
FUNCTION DisposeUserData    (theUserData: UserData): OSErr;
```

```
FUNCTION PutUserDataIntoHandle
                            (theUserData UserData; h Handle): OSErr;
FUNCTION NewUserDataFromHandle
                            (h Handle; VAR theUserData: UserData): OSErr;
```

Routines for Editing Movies

Editing Movies

```
FUNCTION PutMovieOnScrap    (theMovie: Movie;
                             movieScrapFlags: LongInt): OSErr;
FUNCTION NewMovieFromScrap  (newMovieFlags: LongInt): Movie;
PROCEDURE SetMovieSelection
                            (theMovie: Movie; selectionTime: TimeValue;
                             selectionDuration: TimeValue);
PROCEDURE GetMovieSelection
                            (theMovie: Movie; VAR selectionTime: TimeValue;
                             VAR selectionDuration: TimeValue);
FUNCTION CutMovieSelection  (theMovie: Movie): Movie;
FUNCTION CopyMovieSelection
                            (theMovie: Movie): Movie;
PROCEDURE PasteMovieSelection
                            (theMovie: Movie; src: Movie);
PROCEDURE AddMovieSelection
                            (theMovie: Movie; src: Movie);
PROCEDURE ClearMovieSelection
                            (theMovie: Movie);
FUNCTION IsScrapMovie       (targetTrack: Track): Component;
FUNCTION PasteHandleIntoMovie
                            (h: Handle; handleType: OSType;
                             theMovie: Movie; flags: LongInt;
                             userComp: ComponentInstance): OSErr;
FUNCTION PutMovieIntoTypedHandle
                            (theMovie: Movie; targetTrack: Track;
                             handleType: OSType; publicMovie: Handle;
                             start: TimeValue; dur: TimeValue;
                             flags: long; userComp: ComponentInstance):
                             OSErr;
```

Undo for Movies

```
FUNCTION NewMovieEditState  (theMovie: Movie): MovieEditState;
FUNCTION UseMovieEditState  (theMovie: Movie; toState: MovieEditState):
                             OSErr;
FUNCTION DisposeMovieEditState
                           (state: MovieEditState): OSErr;
```

Low-Level Movie-Editing Routines

```
FUNCTION InsertMovieSegment (srcMovie: Movie; dstMovie: Movie;
                             srcIn: TimeValue; srcDuration: TimeValue;
                             dstIn: TimeValue): OSErr;
FUNCTION InsertEmptyMovieSegment
                           (dstMovie: Movie; dstIn: TimeValue;
                            dstDuration: TimeValue): OSErr;
FUNCTION DeleteMovieSegment
                           (theMovie: Movie; in: TimeValue;
                            duration: TimeValue): OSErr;
FUNCTION ScaleMovieSegment  (theMovie: Movie; in: TimeValue;
                             oldDuration: TimeValue;
                             newDuration: TimeValue): OSErr;
FUNCTION CopyMovieSettings  (srcMovie: Movie; dstMovie: Movie): OSErr;
```

Editing Tracks

```
FUNCTION InsertTrackSegment
                           (srcTrack: Track; dstTrack: Track;
                            srcIn: TimeValue; srcDuration: TimeValue;
                            dstIn: TimeValue): OSErr;
FUNCTION InsertEmptyTrackSegment
                           (dstTrack: Track; dstIn: TimeValue;
                            dstDuration: TimeValue): OSErr;
FUNCTION InsertMediaIntoTrack
                           (theTrack: Track; trackStart: TimeValue;
                            mediaTime: TimeValue; mediaDuration: TimeValue;
                            mediaRate: Fixed): OSErr;
FUNCTION DeleteTrackSegment
                           (theTrack: Track; in: TimeValue;
                            duration: TimeValue): OSErr;
```

```
FUNCTION ScaleTrackSegment  (theTrack: Track; in: TimeValue;
                             oldDuration: TimeValue;
                             newDuration: TimeValue): OSErr;
FUNCTION CopyTrackSettings  (srcTrack: Track; dstTrack: Track): OSErr;
FUNCTION GetTrackEditRate   (theTrack: Track; atTime: TimeValue): Fixed;
```

Undo for Tracks

```
FUNCTION NewTrackEditState  (theTrack: Track): TrackEditState;
FUNCTION UseTrackEditState  (theTrack: Track; state: TrackEditState): OSErr;
FUNCTION DisposeTrackEditState
                            (state: TrackEditState): OSErr;
```

Adding Samples to Media Structures

```
FUNCTION BeginMediaEdits    (theMedia: Media): OSErr;
FUNCTION EndMediaEdits      (theMedia: Media): OSErr;
FUNCTION AddMediaSample     (theMedia: Media; dataIn: Handle;
                             inOffset: LongInt; size: LongInt;
                             durationPerSample: TimeValue;
                             sampleDescriptionH: SampleDescriptionHandle;
                             numberOfSamples: LongInt;
                             sampleFlags: Integer;
                             VAR sampleTime: TimeValue): OSErr;
FUNCTION AddMediaSampleReference
                            (theMedia: Media; dataOffset: LongInt;
                             size: LongInt; durationPerSample: TimeValue;
                             sampleDescriptionH: SampleDescriptionHandle;
                             numberOfSamples: LongInt;
                             sampleFlags: Integer;
                             VAR sampleTime: TimeValue): OSErr;
FUNCTION GetMediaSample     (theMedia: Media; dataOut: Handle;
                             maxSizeToGrow: LongInt; VAR size: LongInt;
                             time: TimeValue;
                             VAR sampleTime: TimeValue;
                             VAR durationPerSample: TimeValue;
                             sampleDescriptionH: SampleDescriptionHandle;
                             VAR sampleDescriptionIndex: LongInt;
                             maxNumberOfSamples: LongInt;
                             VAR numberOfSamples: LongInt;
                             VAR sampleFlags: Integer): OSErr;
```

```
FUNCTION GetMediaSampleReference
                            (theMedia: Media; VAR dataOffset: LongInt;
                             VAR size: LongInt; time: TimeValue;
                             VAR sampleTime: TimeValue;
                             VAR durationPerSample: TimeValue;
                             sampleDescriptionH: SampleDescriptionHandle;
                             VAR sampleDescriptionIndex: LongInt;
                             maxNumberOfSamples: LongInt;
                             VAR numberOfSamples: LongInt;
                             VAR sampleFlags: Integer): OSErr;
```

Media Routines

Selecting Media Handlers

```
PROCEDURE GetMediaHandlerDescription
                            (theMedia: Media; VAR mediaType: OSType;
                             VAR creatorName: Str255;
                             VAR creatorManufacturer: OSType);
FUNCTION GetMediaHandler    (theMedia: Media): MediaHandler;
FUNCTION SetMediaHandler    (theMedia: Media;
                             mH: MediaHandlerComponent): OSErr;
PROCEDURE GetMediaDataHandlerDescription
                            (theMedia: Media; index: Integer;
                             VAR dhType: OSType;
                             VAR creatorName: Str255;
                             VAR creatorManufacturer: OSType);
FUNCTION GetMediaDataHandler
                            (theMedia: Media; index: Integer): DataHandler;
FUNCTION SetMediaDataHandler
                            (theMedia: Media; index: Integer;
                             dataHandler: DataHandlerComponent): OSErr;
```

Video Media Handler Routines

```
FUNCTION SetVideoMediaGraphicsMode
                            (mh: MediaHandler; graphicsMode: LongInt;
                             opColor: RGBColor): HandlerError;
FUNCTION GetVideoMediaGraphicsMode
                            (mh: MediaHandler; VAR graphicsMode: LongInt;
                             VAR opColor: RGBColor): HandlerError;
```

Sound Media Handler Routines

```
FUNCTION SetSoundMediaBalance
                            (mh: MediaHandler;
                             balance: Integer): HandlerError;
FUNCTION GetSoundMediaBalance
                            (mh: MediaHandler;
                             VAR balance: Integer): HandlerError;
```

Text Media Handler Routines

```
FUNCTION AddTextSample      (mh: MediaHandler; text: Ptr; size: LongInt;
                             fontNumber: Integer; fontSize: Integer;
                             textFace: Style; textColor: RGBColor;
                             backColor: RGBColor;
                             textJustification: Integer; VAR textBox: Rect;
                             displayFlags: LongInt;
                             scrollDelay: TimeValue;
                             hiliteStart: Integer; hiliteEnd: Integer;
                             VAR rgbColor: RGBColor;
                             duration: TimeValue;
                             VAR sampleTime: TimeValue): ComponentResult;
FUNCTION AddTESample        (mh: MediaHandler; hTE: TEHandle;
                             VAR backColor: RGBColor;
                             textJustification: Integer; VAR textBox: Rect;
                             displayFlags: LongInt;
                             scrollDelay: TimeValue;
                             hiliteStart: Integer; hiliteEnd: Integer;
                             VAR rgbColor: RGBColor;
                             duration: TimeValue;
                             VAR sampleTime: TimeValue): ComponentResult;
FUNCTION AddHiliteSample    (mh: MediaHandler; hiliteStart: Integer;
                             hiliteEnd: Integer; VAR rgbColor: RGBColor;
                             duration: TimeValue;
                             VAR sampleTime: TimeValue): ComponentResult;
FUNCTION FindNextText       (mh: MediaHandler; text: Ptr; size: LongInt;
                             findFlags: Integer; startTime: TimeValue;
                             VAR foundTime: TimeValue;
                             VAR foundDuration: TimeValue;
                             VAR offset: LongInt): ComponentResult;
FUNCTION HiliteTextSample   (mh: MediaHandler; sampleTime: TimeValue;
                             hiliteStart: Integer;
                             hiliteEnd: Integer;
                             VAR rgbHiliteColor: RGBColor): ComponentResult;
FUNCTION SetTextProc        (mh: MediaHandler; TextProc: ProcPtr;
                             refcon: LongInt): ComponentResult;
```

Routines for Creating File Previews

```
FUNCTION MakeFilePreview    (resRefNum: Integer;
                             progress: ProgressProcRecordPtr): OSErr;
FUNCTION AddFilePreview     (resRefNum: Integer; previewType: OSType;
                             previewData: Handle): OSErr;
```

Routines for Displaying File Previews

```
PROCEDURE SFGetFilePreview  (where: Point; prompt: Str255;
                             fileFilter: FileFilterProcPtr;
                             numTypes: Integer; typeList: SFTypeList;
                             dlgHook: DlgHookProcPtr; VAR reply: SFReply);
PROCEDURE SFPGetFilePreview
                            (where: Point; prompt: Str255;
                             fileFilter: FileFilterProcPtr;
                             numTypes: Integer; typeList: SFTypeList;
                             dlgHook: DlgHookProcPtr; VAR reply: SFReply;
                             dlgID: Integer; filterProc:
                             ModalFilterProcPtr);
PROCEDURE StandardGetFilePreview
                           (fileFilter: FileFilterProcPtr;
                             numTypes: Integer; typeList: SFTypeList;
                             VAR reply: StandardFileReply);
PROCEDURE CustomGetFilePreview
                           (fileFilter: FileFilterYDProcPtr;
                             numTypes: Integer; typeList: SFTypeList;
                             VAR reply: StandardFileReply; dlgID: Integer;
                             where: Point; dlgHook: DlgHookYDProcPtr;
                             filterProc: ModalFilterYDProcPtr;
                             activeList: Ptr;
                             activateProc: ActivateYDProcPtr;
                             yourDataPtr: UNIV Ptr);
```

Time Base Routines

Creating and Disposing of Time Bases

```
FUNCTION NewTimeBase:       TimeBase;
PROCEDURE DisposeTimeBase   (tb: TimeBase);
PROCEDURE SetMovieMasterClock
                           (theMovie: Movie; clockMeister: Component;
                             slaveZero: TimeRecord);
```

```
PROCEDURE SetMovieMasterTimeBase
                            (theMovie: Movie; tb: TimeBase;
                             slaveZero: TimeRecord);
PROCEDURE SetTimeBaseMasterClock
                            (slave: TimeBase; clockMeister: Component;
                             slaveZero: TimeRecord);
FUNCTION GetTimeBaseMasterClock
                            (tb: TimeBase): ComponentInstance;
PROCEDURE SetTimeBaseMasterTimeBase
                            (slave: TimeBase; master: TimeBase;
                             slaveZero: TimeRecord);
FUNCTION GetTimeBaseMasterTimeBase
                            (tb: TimeBase): TimeBase;
PROCEDURE SetTimeBaseZero   (tb: TimeBase; VAR zero: TimeRecord);
```

Working With Time Base Values

```
PROCEDURE SetTimeBaseTime   (tb: TimeBase; tr: TimeRecord);
PROCEDURE SetTimeBaseValue  (tb: TimeBase; t: TimeValue; s: TimeScale);
FUNCTION GetTimeBaseTime    (tb: TimeBase; s: TimeScale;
                             VAR tr: TimeRecord): TimeValue;
PROCEDURE SetTimeBaseRate   (tb: TimeBase; r: Fixed);
FUNCTION GetTimeBaseRate    (tb: TimeBase): Fixed;
FUNCTION GetTimeBaseEffectiveRate
                            (tb: TimeBase): Fixed;
PROCEDURE SetTimeBaseStartTime
                            (tb: TimeBase; VAR tr: TimeRecord);
FUNCTION GetTimeBaseStartTime
                            (tb: TimeBase; s: TimeScale;
                             tr: TimeRecord): TimeValue;
PROCEDURE SetTimeBaseStopTime
                            (tb: TimeBase; VAR tr: TimeRecord);
FUNCTION GetTimeBaseStopTime
                            (tb: TimeBase; s: TimeScale;
                             tr: TimeRecord): TimeValue;
PROCEDURE SetTimeBaseFlags  (tb: TimeBase; timeBaseFlags: LongInt);
FUNCTION GetTimeBaseFlags   (tb: TimeBase): LongInt;
FUNCTION GetTimeBaseStatus  (tb: TimeBase;
                             VAR unpinnedTime: TimeRecord): LongInt;
```

Working With Times

```
PROCEDURE AddTime           (VAR dst: TimeRecord; src: TimeRecord);
PROCEDURE SubtractTime      (VAR dst: TimeRecord; src: TimeRecord);
PROCEDURE ConvertTime       (VAR inout: TimeRecord; newBase: TimeBase);
PROCEDURE ConvertTimeScale  (VAR inout: TimeRecord; newScale: TimeScale);
```

Time Base Callback Routines

```
FUNCTION NewCallBack        (tb: TimeBase; cbType: Integer): QTCallBack;
FUNCTION CallMeWhen         (cb: QTCallBack; callBackProc: QTCallBackProc;
                             refcon: LongInt; param1: LongInt;
                             param2: LongInt; param3: LongInt): OSErr;
PROCEDURE CancelCallBack    (cb: QTCallBack);
PROCEDURE DisposeCallBack   (cb: QTCallBack);
FUNCTION GetCallBackTimeBase
                            (cb: QTCallBack): TimeBase;
FUNCTION GetCallBackType    (cb: QTCallBack): Integer;
```

Matrix Routines

```
PROCEDURE SetIdentityMatrix
                            (VAR matrix: MatrixRecord);
FUNCTION GetMatrixType      (m: MatrixRecord): Integer;
PROCEDURE CopyMatrix        (m1: MatrixRecord; VAR m2: MatrixRecord);
FUNCTION EqualMatrix        (m1: MatrixRecord; m2: MatrixRecord): Boolean;
PROCEDURE TranslateMatrix   (VAR m: MatrixRecord; deltaH: Fixed;
                             deltaV: Fixed);
PROCEDURE ScaleMatrix       (VAR m: MatrixRecord; scaleX: Fixed;
                             scaleY: Fixed; aboutX: Fixed; aboutY: Fixed);
PROCEDURE RotateMatrix      (VAR m: MatrixRecord; degrees: Fixed;
                             aboutX: Fixed; aboutY: Fixed);
PROCEDURE SkewMatrix        (VAR m: MatrixRecord; skewX: Fixed;
                             skewY: Fixed; aboutX: Fixed; aboutY: Fixed);
PROCEDURE ConcatMatrix      (a: MatrixRecord; VAR b: MatrixRecord);
FUNCTION InverseMatrix      (m: MatrixRecord;
                             VAR im: MatrixRecord): Boolean;
FUNCTION TransformPoints    (mp: MatrixRecord; VAR pt1: Point;
                             count: LongInt): OSErr;
FUNCTION TransformFixedPoints
                            (m: MatrixRecord; VAR fpt: FixedPoint;
                             count: LongInt): OSErr;
```

```
FUNCTION TransformRect      (m: MatrixRecord; VAR r: Rect;
                             VAR fpp: FixedPoint): Boolean;
FUNCTION TransformFixedRect (m: MatrixRecord; VAR fr: FixedRect;
                             VAR fpp: FixedPoint): Boolean;
FUNCTION TransformRgn       (mp: MatrixRecord; r: RgnHandle): OSErr;
PROCEDURE RectMatrix        (VAR matrix: MatrixRecord; srcRect: Rect;
                             dstRect: Rect);
PROCEDURE MapMatrix         (VAR matrix: MatrixRecord; fromRect: Rect;
                             toRect: Rect);
```

Application-Defined Routines

Progress Routines

```
FUNCTION MyProgressProc     (theMovie: Movie; message: Integer;
                             whatOperation: Integer;
                             percentDone: Fixed; refcon: LongInt): OSErr;
```

Cover Routines

```
FUNCTION MyCoverProc        (theMovie: Movie; changedRgn: RgnHandle;
                             refcon: LongInt): OSErr;
```

Error-Notification Routines

```
PROCEDURE MyErrProc         (theErr: OSErr; refcon: LongInt);
```

Movie Callout Routines

```
FUNCTION MyCallOutProc      (refcon: long): Boolean;
```

File Filter Routines

```
FUNCTION MyFileFilter       (paramBlock: ParmBlkPtr): Boolean;
```

Custom Routines

```
FUNCTION MyDlgHook          (item: Integer; theDialog: DialogPtr;
                             myDataPtr: Ptr): Integer;
```

Modal-Dialog Filter Routines

```
FUNCTION MyModalFilter      (theDialog: DialogPtr;
                             VAR theEvent: EventRecord; itemHit: Integer;
                             myDataPtr: Ptr): Boolean;
```

Standard File Activation Routines

```
PROCEDURE MyActivateProc    (theDialog: DialogPtr; itemNo: Integer;
                             activating: Boolean; myDataPtr: Ptr);
```

Callback Event Routines

```
PROCEDURE MyCallBackProc    (cb: QTCallBack; refcon: LongInt);
```

Text Routines

```
PROCEDURE MyTextProc        (theText: Handle; theMovie: Movie;
                             VAR displayFlag: Integer; refcon: LongInt);
```

Result Codes

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
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
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

CHAPTER 3

Image Compression Manager

Contents

This chapter describes the Image Compression Manager. The Image Compression Manager provides image-compression and image-decompression services to applications and other managers. If you are developing an application that works with images, you should read this chapter to familiarize yourself with the features of the Image Compression Manager. If you want to develop a compressor or decompressor for use on the Macintosh computer, see *Inside Macintosh: QuickTime Components* for information about the software interfaces that your component must support in order to work with the Image Compression Manager.

Image compression benefits you by decreasing the amount of storage required for image data, decreasing the time required to exchange image data across networks, and decreasing the time required to read data from disks and CD-ROM volumes.

This chapter is divided into the following major sections:

- "Introduction to the Image Compression Manager" contains a general introduction to the features provided by the Image Compression Manager.
- "About Image Compression" presents background information on image compression and image-compression algorithms, and it describes the features of the image compressors and decompressors supplied by Apple.
- "Using the Image Compression Manager" discusses how you can use the features of the Image Compression Manager to compress and decompress still images and image sequences—within this section are a number of shorter sections that discuss more advanced topics, including key frames, fast dithering, and compressor and decompressor components.
- "Image Compression Manager Reference" describes the data types and functions provided by the Image Compression Manager.
- "Summary of the Image Compression Manager" contains a condensed listing of the constants, data types, and functions provided by the Image Compression Manager in C and in Pascal.

Introduction to the Image Compression Manager

The Image Compression Manager provides your application with an interface for compressing and decompressing images and sequences of images that is independent of devices and algorithms.

Uncompressed image data requires a large amount of storage space. Storing a single 640-by-480 pixel image in 32-bit color can require as much as 1.2 MB. Sequences of images, like those that might be contained in a QuickTime movie, demand substantially more storage than single images. This is true even for sequences that consist of fairly small images, because the movie consists of such a large number of those images. Consequently, minimizing the storage requirements for image data is an important consideration for any application that works with images or sequences of images.

The Image Compression Manager allows your application to

- use a common interface for all image-compression and image-decompression operations
- take advantage of any compression software or hardware that may be present in a given Macintosh configuration
- store compressed image data in pictures
- temporally compress sequences of images, further reducing the storage requirements of movies
- display compressed PICT files without the need to modify your application
- use an interface that is appropriate for your application—a high-level interface if you do not need to manipulate many compression parameters or a low-level interface that provides you greater control over the compression operation

The Image Compression Manager compresses images by invoking **image compressor components** and decompresses images using **image decompressor components.** Compressor and decompressor components are code resources that present a standard interface to the Image Compression Manager and provide image-compression and image-decompression services, respectively. The Image Compression Manager receives application requests and coordinates the actions of the appropriate components. The components perform the actual compression and decompression. Compressor and decompressor components are standard components and are managed by the Component Manager. For detailed information about creating compressor and decompressor components, see *Inside Macintosh: QuickTime Components.*

Because the Image Compression Manager is independent of specific compression algorithms and drivers, it provides a number of advantages to developers of image-compression algorithms. Specifically, compressor and decompressor components can

- present a common application interface for software-based compressors and hardware-based compressors
- provide several different compressors and compression options, allowing the Image Compression Manager or the application to choose the appropriate tool for a particular situation

Data That Is Suitable for Compression

One way to represent an image is with a pixel map, which stores a color for every pixel. For most images, however, a pixel map is an inefficient storage format. For example, a pixel map containing a solid black image would contain the color black stored over and over and over again. By compressing the image, some of this redundant information can be eliminated. The compressed image can occupy much less storage than a pixel map and can be decompressed to a pixel map when necessary.

In addition, human perception of visual images exhibits special qualities that can be exploited to further compress image data. Image-compression algorithms take advantage of these properties to reduce the amount of information required to describe an image well enough to allow a person to see it.

A **lossless compression** technique can recreate an exact copy of the original image from the compressed form. Small changes in the image are not objectionable in most applications, however, so most compressors sacrifice some accuracy in order to further decrease the size of the compressed data. However, the compressor carefully chooses the data to omit so that the human visual system compensates for the loss and fools the user into seeing what appears to be the original image.

The Image Compression Manager works only with image data. The Image Compression Manager is primarily useful for compressing pictures that have pixel map images, such as those obtained from scanned still images or digitized video images, or from paint or three-dimensional rendering applications. You do not achieve significant compression treating pictures that are stored as groups of graphics primitives, such as those created by drawing, computer-aided design (CAD), or three-dimensional modeling applications. These applications create images in a compact format that precisely states the characteristics of the objects in the image. In fact, if you were to convert such images to pixel map representations and then compress the resulting image with the Image Compression Manager, you would probably end up with a larger, less precise image than the original. If a picture contains both primitives and pixel map image data (such as text or lines drawn over a painted or digitized image) the Image Compression Manager compresses the pixel map data and leaves the graphics primitives unchanged.

The Image Compression Manager also provides services for compressing and decompressing sequences of images or **frames** (another term for a single visual image in an image sequence). When processing a sequence, compressors may perform **temporal compression**, compressing the sequence by eliminating information that is redundant from one frame to the next. This temporal compression differs from **spatial compression,** which is performed on individual images or frames within a sequence. You may use both techniques on a single sequence.

Compressor components perform temporal compression by comparing the current frame in a sequence with the previous frame. The compressor then stores information about only those pixels that change significantly between the two images. When adjacent images contain substantially similar visual information, as is often the case in movies, temporal compression can significantly reduce the amount of data required to describe the images in the sequence. Your application indicates the desired quality level for the compressed image. The compressor uses this value to govern the extent to which it takes advantage of temporal redundancy between images. There is also a spatial quality level that you can use to control the amount of spatial compression applied to each individual image. Both of these quality values govern the amount of accuracy that is lost in the compressed image.

Note that the Image Compression Manager does not maintain any time information for an image sequence. Rather, the Image Compression Manager maintains the order and content of the images in the sequence while the Movie Toolbox handles all timing considerations.

Storing Images

The Image Compression Manager can compress two kinds of image data: pictures and pixel maps. Pictures may be stored in memory, in a resource, or in a PICT file. Pixel maps are normally stored in a window or offscreen buffer. When compressing an image from a PICT file, the Image Compression Manager provides facilities that allow applications to spool data to and from the disk file, as appropriate to the operation. These application-provided data-loading and data-unloading functions allow arbitrarily large images to be compressed or decompressed without requiring large amounts of memory.

Applications must convert images that are not stored as pictures or pixel maps into one of these formats before compressing them. The Image Compression Manager contains several high-level functions that make it quite easy for applications to work with compressed images that are stored as PICT files. See “Working With Pictures” on page 3-24 for more information.

About Image Compression

This section provides some background information regarding image compression. This discussion has been divided into two main sections. The first, “Image-Compression Characteristics,” describes the key features you can use to choose a compression algorithm for your image data. The second, “Compressors Supplied by Apple,” discusses the compressors that are supplied with the Image Compression Manager by Apple.

Image-Compression Characteristics

There are three main characteristics by which you can judge image-compression algorithms: compression ratio, compression speed, and image quality. You can use these characteristics to determine the suitability of a given compression algorithm to your application. The following paragraphs discuss each of these attributes in more detail.

Compression Ratio

The compression ratio is equal to the size of the original image divided by the size of the compressed image. This ratio gives an indication of how much compression is achieved for a particular image.

The compression ratio achieved usually indicates the picture quality. Generally, the higher the compression ratio, the poorer the quality of the resulting image. The trade-off between compression ratio and picture quality is an important one to consider when compressing images.

Furthermore, some compression schemes produce compression ratios that are highly dependent on the image content. This aspect of compression is called **data dependency**. Using an algorithm with a high degree of data dependency, an image of a crowd at a football game (which contains a lot of detail) may produce a very small compression ratio, whereas an image of a blue sky (which consists mostly of constant colors and intensities) may produce a very high compression ratio.

Compression Speed

Compression time and decompression time are defined as the amount of time required to compress and decompress a picture, respectively. Their value depends on the following considerations:

- the complexity of the compression algorithm
- the efficiency of the software or hardware implementation of the algorithm
- the speed of the utilized processor or auxiliary hardware

Generally, the faster that both operations can be performed, the better. Fast compression time increases the speed with which material can be created. Fast decompression time increases the speed with which the user can display and interact with images.

Image Quality

Image quality describes the fidelity with which an image-compression scheme recreates the source image data. Compression schemes can be characterized as being either lossy or lossless. Lossless schemes preserve all of the original data. **Lossy compression** does not preserve the data precisely; image data is lost, and it cannot be recovered after compression. Most lossy schemes try to compress the data as much as possible, without decreasing the image quality in a noticeable way. Some schemes may be either lossy or lossless, depending upon the quality level desired by the user.

Compressors Supplied by Apple

Apple supplies six image-compression algorithms with the Image Compression Manager. This section discusses each of these compressors and identifies their strengths and weaknesses in light of the compression characteristics just discussed. You can use this discussion as a guideline for choosing a compression algorithm for your specific situation. All the compressors support both temporal and spatial compression except for the Photo and Raw Compressors, which support only spatial compression.

The Photo Compressor

The Photo Compressor implements the **Joint Photographic Experts Group** (JPEG) algorithm for image compression. JPEG is an international standard for compressing still images. The version of JPEG supplied with QuickTime complies with the baseline International Standards Organization (ISO) standard bitstream, version 9R9.

The Photo Compressor performs best on images that vary smoothly or that do not have a large percentage of their areas devoted to edges or other types of sharp detail. This is the case for most natural (that is, nonsynthetic) images. In practice, you will find that compression ratios are highly dependent on source images, but they generally range from 5:1 to 50:1 at 24 bits per pixel, with good picture quality resulting from compression ratios between 10:1 and 20:1.

Picture quality is generally very good to excellent and is often good enough for use in demanding desktop publishing applications. Very high-resolution images obtained through the use of 24-bit color scanners would best be compressed using the Photo Compressor. This compressor is good for 8-bit grayscale images; it is not well suited to 1-bit images or non-natural images that usually have high contrast.

On a Macintosh IIsi, the Photo Compressor can compress a 24-bit, 640-by-480 pixel image at a normal quality setting in 7.5 seconds, achieving a compression ratio of 10:1. Decompressing the same image takes 6.5 seconds.

The Video Compressor

The Video Compressor employs an image-compression method developed by Apple. This method was designed to permit very fast decompression times while maintaining reasonably good picture quality. This algorithm's rapid decompression allows applications to display color images or drawings at interactive speeds. This algorithm is best suited for use with sequences of video data.

The Video Compressor is better suited to digitized video content rather than synthetically generated images. This compressor supports both spatial and temporal compression. If you use only spatial compression, you may obtain compression ratios from 5:1 to 8:1 with reasonably good quality at 24-bit pixel depths. If you use both spatial and temporal compression, the compression ratio range extends from 5:1 to 25:1.

On a Macintosh IIsi, the Video Compressor can compress a 24-bit, 640-by-480 pixel image at a normal quality setting in 3.5 seconds, achieving a compression ratio of 6.5:1. Decompressing the same image takes 1.0 second.

The Compact Video Compressor

The Compact Video Compressor is best suited to compressing 16-bit and 24-bit video sequences. It employs a lossy algorithm developed by Apple that is highly asymmetrical. In other words, it takes significantly longer to compress a frame than it does to decompress that frame. Compressing a 24-bit, 640-by-480 image on a Macintosh IIsi computer takes approximately 2.5 minutes, achieving a compression ratio of 18.5:1. Decompressing the image takes less than a second.

Compared to the Video Compressor, the Compact Video Compressor obtains higher compression ratios, better image quality, and faster playback speeds. The Compact Video Compressor can constrain data rates to user-definable levels. This is particularly important when compressing material for playback from CD-ROM discs.

For best quality results, the Compact Video Compressor should be used on raw source data that has not been compressed with a highly lossy compressor—such as the Video Compressor.

The Animation Compressor

The Animation Compressor employs a compression algorithm developed by Apple. This technique is best suited to animation and computer-generated video content. In addition, the Animation Compressor can be used to compress sequences of screen images, such as might be generated for a training application.

The Animation Compressor stores images in run-length encoded format, and it can work in either a lossy or a lossless mode. The lossless mode maintains picture content precisely, storing an animation as a series of run-length encoded images. The lossy mode loses some image quality.

The Animation Compressor's performance and achieved compression ratios are highly dependent on the type of images in a scene. The Animation Compressor is very sensitive to picture changes, and it works best on a clean image that has been generated synthetically. Images captured from videotape generally have considerable visual noise, which can corrupt the inherent similarity of the pixels and make it more difficult for the Animation Compressor to achieve good compression. This compressor works at all pixel depths.

On a Macintosh IIsi, the Animation Compressor can compress a 24-bit, 640-by-480 pixel image at a normal quality setting in 2.0 seconds, achieving a compression ratio of 1.3:1. Decompressing the same image takes 1 second.

The Graphics Compressor

The Graphics Compressor employs a compression algorithm developed by Apple. This compressor is best suited to 8-bit still images and image sequences in applications where compression ratio is more important than decompression speed.

The Graphics Compressor is a good alternative to the Animation Compressor whenever performance is less important than compression ratio. In general, the Graphics Compressor generates a compressed image that is one-half the size of the same image compressed by the Animation Compressor. However, the Graphics Compressor can decompress the image at only half the speed of the Animation Compressor. Therefore, you should consider using the Graphics Compressor with relatively slow storage devices, such as CD-ROM discs. In these circumstances, the Graphics Compressor has sufficient time to decompress the image or image sequence.

On a Macintosh IIsi, the Graphics Compressor can compress a 640-by-480 pixel image that has been dithered to 8-bit pixel depth at a normal quality setting in 6.5 seconds, achieving a compression ratio of 2.5:1. Decompressing the same image takes 1.0 second.

The Raw Compressor

The Raw Compressor can reduce image storage requirements by converting an image from one pixel depth to another. For example, converting a 32-bit image to 16-bit format achieves a 2:1 compression ratio. The Raw Compressor can also convert a 32-bit image to 24-bit format by dropping the pad byte. This achieves a 4:3 compression with no loss of quality. The Raw Compressor accomplishes this conversion quickly, and the resulting image retains excellent image quality in most cases.

The Image Compression Manager often uses the Raw Compressor to extend the capabilities of other compressors. For example, the Photo Compressor works directly with only 32-bit color images and 8-bit grayscale images. For color images, the Image Compression Manager uses the Raw Compressor to convert the pixel depth of the original image to 32-bit color or to convert the 32-bit decompressed image to another pixel depth for display.

Image quality can deteriorate when the pixel depth is reduced; however, this technique is generally lossless when converting from a lower pixel depth to a higher depth. With 1, 2, 4, 8, and 24-bit images, the Raw Compressor allows colors to be mapped through a color table.

Note that the resulting image may be larger than the corresponding pixel image in PICT format, because QuickDraw stores PICT images in a run-length encoded format.

Note

These uncompressed QuickTime-specific PICT images cannot be used without QuickTime. u

Performance figures for the Raw Compressor are dependent upon the source and destination pixel depths. (The Raw Compressor is signified by the None option in the standard compression dialog box.)

Types of Images Suitable for Different Compressors

This section presents a series of graphs that indicate the amount of compression you can obtain when you compress still images with the Apple-supplied QuickTime compressors.

Note
Since some compressors make use of temporal compression, these results cannot be used to directly infer results for compressing image sequences (as in QuickTime movies). u

The different compressors take advantage of different properties of an image to achieve their compression; hence, the type of image being compressed significantly affects the amount of compression achieved, as well as the fidelity of the compressed image to the original.

For this comparison, three images that represent three classes of digital images are used. Figure 3-1 provides a photographic image scanned from a photographic slide. This is a natural image and contains no computer-synthesized characters or graphics elements.

Figure 3-1 24-bit photographic image

Figure 3-2 shows a full-color image created by a three-dimensional graphics rendering program. It does not contain the detail of a natural image, but it is a full-color image that needs significantly more than 256 colors to portray it accurately. It is possible to create such an image with a full-color paint or drawing program as well as from a three-dimensional rendering program. Note also that, if an image created by these means has enough detail, it becomes more like a photographic image. Likewise, a natural image with some overlaid graphics or text may fit more closely into this category than the photographic category depending on the proportions of each type of imagery.

Figure 3-2 24-bit synthetic image

Figure 3-3 is an example of a nondithered simple graphic image with fewer than 256 colors. The image is adequately represented by 8 bits per pixel. This image is also special in that it has large horizontal areas that are all of a single color, which is an important characteristic exploited by several compression algorithms, including the normal PICT packing used by QuickDraw.

Figure 3-3 8-bit graphic image

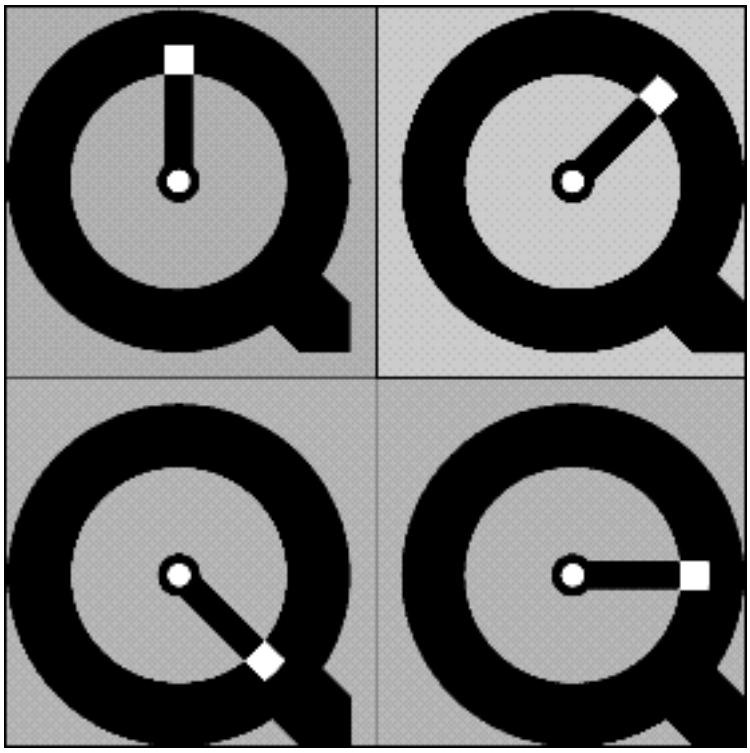


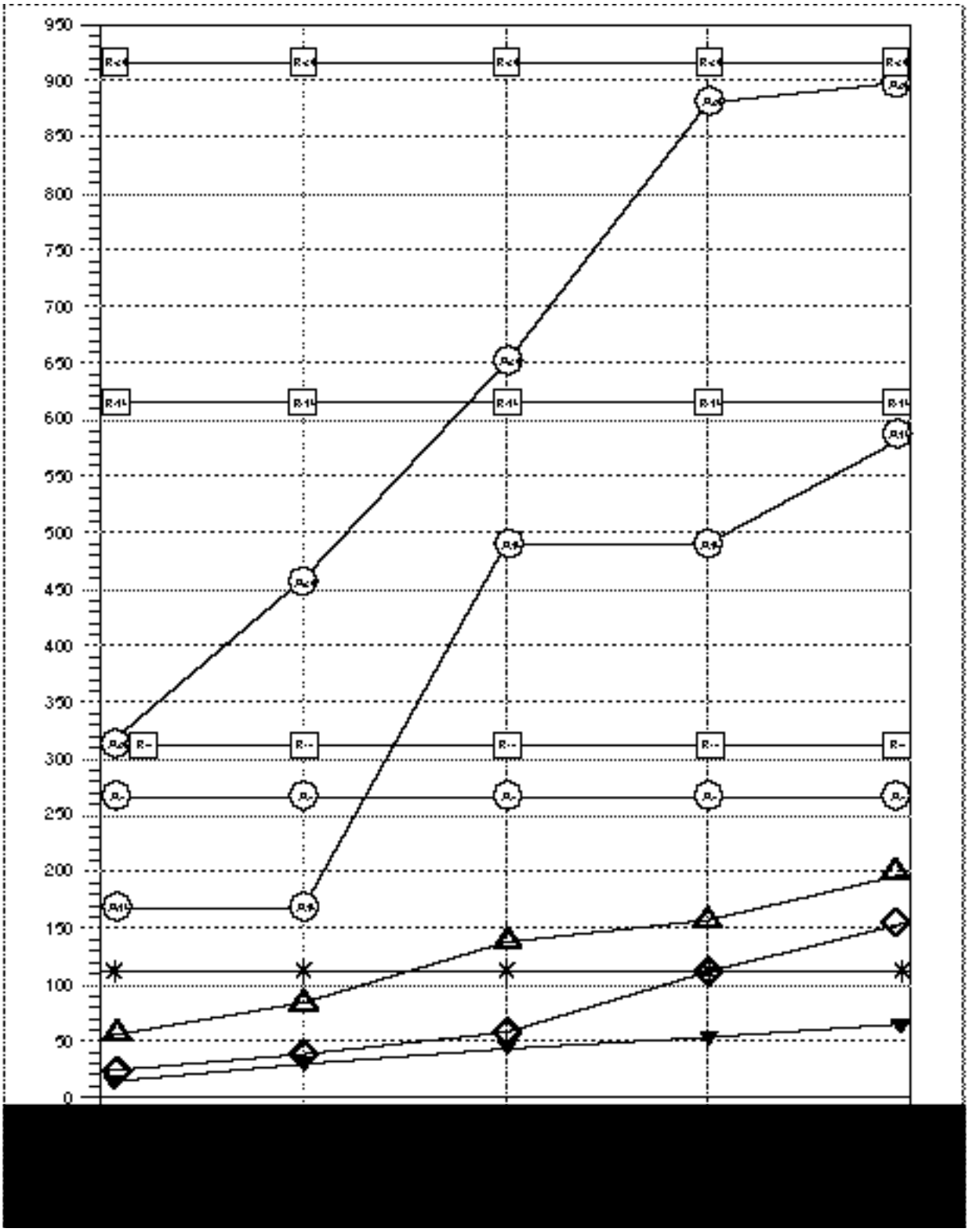
Figure 3-4 is a natural photographic image dithered to 8 bits per pixel.

Figure 3-4 8-bit photographic image

All of the graphs show the compressed data size (in kilobytes) versus the quality of an image at minimum, low, normal, high, and maximum compression settings. The Raw Compressor is included to show the size of the image in raw pixels. The Raw Compressor is not useful for storing still images, since it does not even use the simple packing technique used by QuickDraw (notice that the 24-bit raw format is larger than the uncompressed PICT file).

Figure 3-5 provides a graph that compares compressor performance for the photographic image shown in Figure 3-1. The best compression is obtained by the Compact Video Compressor. The Photo Compressor performs as well as the Compact Video Compressor at minimum, low, and normal compression settings, but does not perform as well at high and maximum settings. However, as you might expect, the Photo Compressor retains the best image quality. The Graphics Compressor stores the image at a smaller size than the highest quality setting of the Photo Compressor, but only stores 256 colors, which significantly degrades the quality of the image. The Video Compressor does almost as well as the Photo Compressor, but the image quality is lower, because of compression artifacts and reduced color resolution. The Animation Compressor retains the color resolution and detail of the image when storing millions of colors and the detail when storing thousands of colors, but it does not achieve nearly as much compression as the other compressors.

Figure 3-5 Compressor performance for a 921 KB, 24-bit, photographic image



The graph in Figure 3-6 compares compressor performance for the full-color, computer-synthesized image shown in Figure 3-2. The Compact Video Compressor again achieves the best overall compression, followed by the Photo and Video Compressors. Again the Graphics Compressor cannot accurately represent all of the colors of the image and is not suitable for use on this type of image. With this image, the Animation Compressor does better than it did with the natural image, and it may be suitable if space constraints are not as important as speed constraints. Because computer-generated images tend to have smoother color gradations than natural images, the loss of color resolution with the Video Compressor and the 16-bit Raw and Animation Compressors is more apparent.

Figure 3-6 Compressor performance for a 502 KB, 24-bit, synthetic image

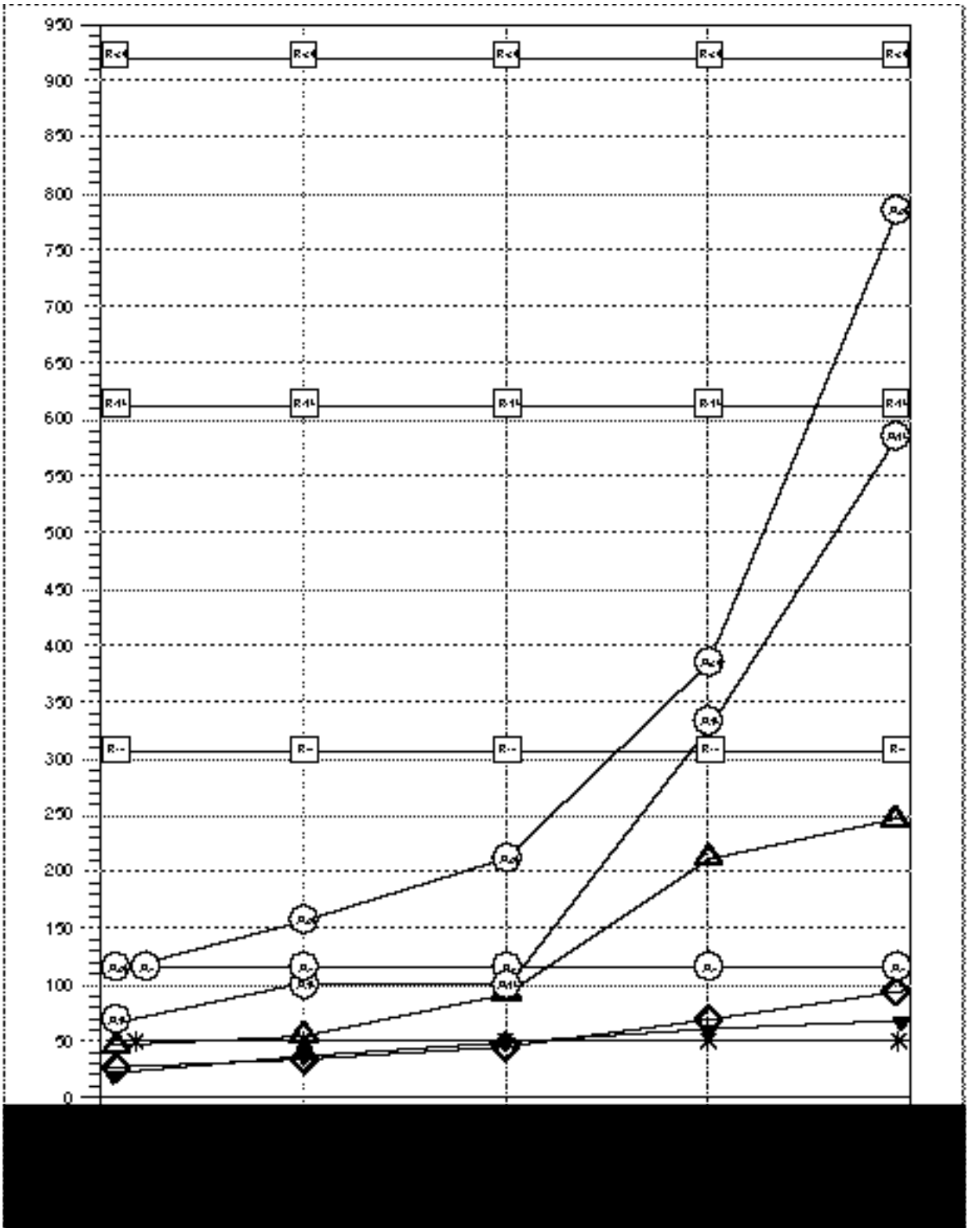
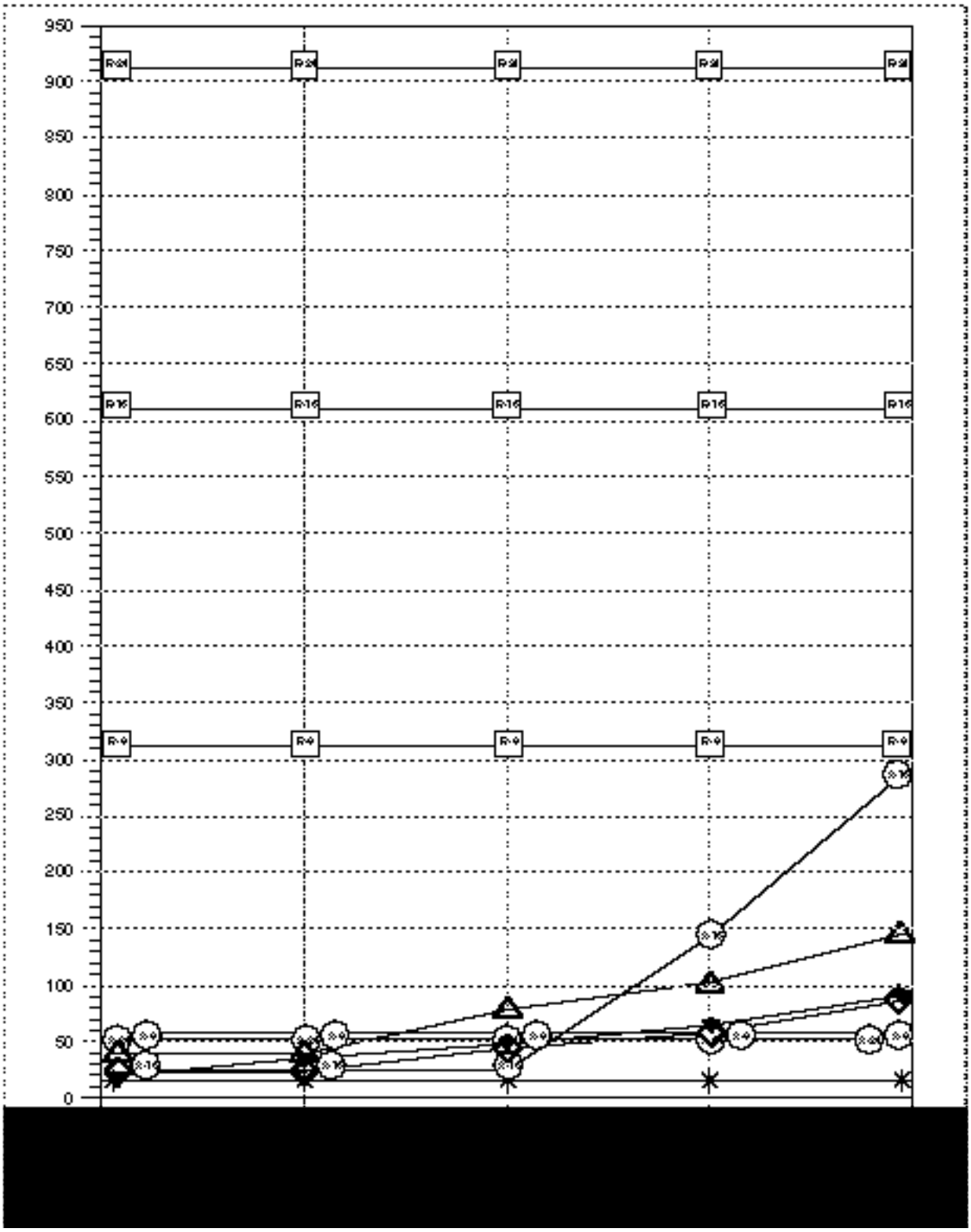


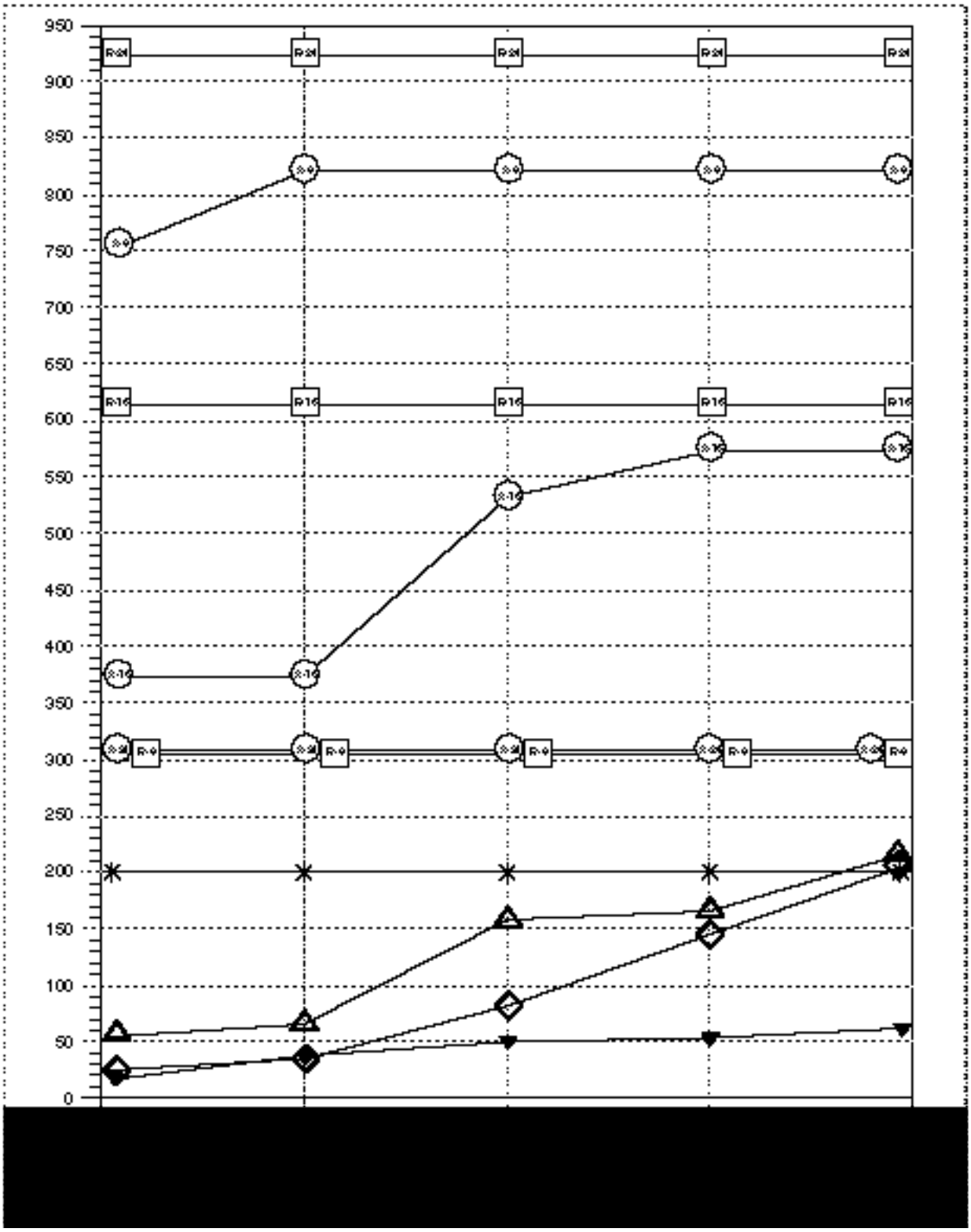
Figure 3-7 compares compressor performance for the simple graphic image shown in Figure 3-3. The Graphics Compressor is the only reasonable choice. Not only does it produce the best compression, but also it stores the image without losing any of the image's detail, since there are fewer than 256 colors in the source image. The Photo and Compact Video Compressors get some compression, but do not store the image as accurately as the Graphics Compressor. The Video Compressor stores the image even less accurately and does not compress the image well at all. The Animation Compressor also does not store the image with complete accuracy at 16 or even 24 bits per pixel, and the resulting files are much larger than the uncompressed PICT. Although the 8-bit Animation Compressor does store the image accurately, it only achieves half as much compression as the Graphics Compressor and its file is also larger than the original PICT.

Figure 3-7 Compressor performance for a 30 KB, 8-bit, graphic image



The graph in Figure 3-8 compares performance for the 8-bit, dithered, photographic image shown in Figure 3-4. The best results are obtained by the Compact Video Compressor. The rest of the results are almost the same as for the full-color, natural image shown in Figure 3-5, but this time the Graphics Compressor stores the image exactly, since it had only 256 colors to start with. The other compressors do almost as well as they did for the full-color, natural image, but the compression for all of them is a bit worse, due to the added artifacts introduced when the image was converted to 8 bits per pixel. The 16-bit and 24-bit versions of the Animation Compressor do not make sense for this image, since their results are always larger than the original PICT. The Photo and Video Compressors still do well on this image, but they do lose some detail that the Graphics Compressor retains. The losses are minor, however, and the sizes approach the size of the Graphics Compressor's image only at high-quality settings, where the losses are negligible.

Figure 3-8 Compressor performance for a 302 KB, 8-bit, dithered, photographic image



Using the Image Compression Manager

This section discusses several of the ways your application may use the Image Compression Manager to compress and decompress images and sequences of images.

Getting Information About Compressors and Compressed Data

Use the Gestalt environmental selector `gestaltCompressionMgr` to determine whether the Image Compression Manager is available. Gestalt returns a 32-bit value indicating the version of the Image Compression Manager that is installed. This return value is formatted in the same way as the value returned by the `CodecManagerVersion` function (described on page 3-62), and it contains the version number specified as an integer value.

```
#define gestaltCompressionMgr 'icmp'
```

The Image Compression Manager provides a number of functions that allow your application to obtain information about the facilities available for image compression or about compressed images. Your application may use some of these functions to select a specific compressor or decompressor for a given operation or to determine how much memory to allocate to receive a decompressed image. In addition, your application may use some of these functions to determine the capabilities of the components that are available on the user's computer system. You can then condition the options your program makes available to the user based on the user's system configuration.
See "Getting Information About Compressor Components," which begins on page 3-62, and "Getting Information About Compressed Data," which begins on page 3-67, for detailed descriptions of these functions.

Working With Pictures

The Image Compression Manager provides a set of functions that allow applications to work easily with compressed pictures stored in version 2 PICT files. These functions constitute a high-level interface to image compression and decompression. Applications that require little control over the compression process may use these functions to display pictures that contain compressed image data.

Existing programs can display (without changes) pictures that contain compressed image data. When the Image Compression Manager is installed on a system, it installs a new `StdPix` graphics function (see page 3-137 for more information on the `StdPix` graphics function). This function handles all requests to display compressed images. Whenever an application issues the standard QuickDraw `DrawPicture` routine

(described in *Inside Macintosh: Imaging*) to display an image that contains compressed image data, the `StdPix` function decompresses the image by invoking the Image Compression Manager. The function then delivers the decompressed image to the application.

The Image Compression Manager also provides a simple mechanism for creating a picture that contains compressed image data. For example, to place an existing compressed image into a picture, your application could open the picture with QuickDraw's `OpenPicture` (or `OpenCPicture`) function and then call the Image Compression Manager's `DecompressImage` function, as if you were going to display the image. The Image Compression Manager places the compressed image and the other data that describe the image into the picture for you.

The Image Compression Manager stores the following information about a compressed picture:

- the image description, which describes the compression format and characteristics of the compressed image data
- the compressed data for the image
- the transfer mode (source copy mode, dither copy mode, and so on)
- the matte pixel map
- the mask region
- the mapping matrix
- the source rectangle of the image

The Image Compression Manager stores this information in the picture as a new PICT opcode (described in the following paragraphs). When an application draws the compressed picture on a Macintosh computer that is running the Image Compression Manager, the `StdPix` function instructs the Image Compression Manager to decompress the image. If an application tries to read a picture file that contains compressed data on a Macintosh that does not have the Image Compression Manager installed, the system ignores the new opcodes and displays a message that indicates that the user needs QuickTime in order to display the compressed image data. The message states "QuickTime™ and a <Codec Name> decompressor are needed to see this picture".

The Color QuickDraw version 2 picture format includes PICT opcodes for compressed and uncompressed QuickTime images. (An opcode is a hexidecimal number that represents drawing commands and the parameters that affect those drawing commands in a picture.) For more information on the version 2 picture format, see the chapter "Color QuickDraw" in *Inside Macintosh: Imaging*.

The PICT opcodes for compressed and uncompressed QuickTime images are

- opcode $8200, which signals a compressed QuickTime image
- opcode $8201, which signals an uncompressed QuickTime image

Table 3-1 gives an overview of the opcode for QuickTime compressed pictures.

Table 3-1 Fields of the PICT opcode for compressed QuickTime images

Field name	Description	Data size (in bytes)
`Opcode`	Compressed picture data	2
`Size`	Size in bytes of data for this opcode	4
`Version`	Version of this opcode	2
`Matrix`	3 by 3 fixed transformation matrix	36
`MatteSize`	Size of matte data in bytes	4
`MatteRect`	Rectangle for matte data	8
`Mode`	Transfer mode	2
`SrcRect`	Rectangle for source	8
`Accuracy`	Preferred accuracy	4
`MaskSize`	Size of mask region in bytes	4

S **WARNING**

Do not attempt to read opcodes directly. For compatibility reasons, use Toolbox routines to access this information. s

The `MaskSize` field of opcode $8200 is followed by five variable fields:

- The matte image description, which contains the image description structure for the matte. The variable size is specified in the first long integer in the opcode. This field is not included if the `MatteSize` field is 0.
- The matte data, which contains the compressed data for the matte. The size of this field is defined by the `MatteSize` field described in Table 3-1. This field is not included if the `MatteSize` field is 0.
- The mask region, which contains the region for masking. The size of this variable is defined by the `MaskSize` field described in Table 3-1. This field is not included if the `MaskSize` field is 0.
- The image description structure for this data. The size of this variable is specified in the first long integer in the `idSize` field of this image description.
- The image data, which contains the compressed data for the image. The size of the image data is specified in the image description's `dataSize` field.

See "The Image Description Structure" beginning on page 3-49 for details on the `idSize` and `dataSize` fields.

Table 3-2 provides an overview of the structure of uncompressed QuickTime images.

Table 3-2 Fields of the PICT opcode for uncompressed QuickTime images

Field name	Description	Data size (in bytes)
`Opcode`	Uncompressed picture data	2
`Size`	Size in bytes of data for this opcode	4
`Version`	Version of this opcode	2
`Matrix`	3 by 3 fixed transformation matrix	36
`MatteSize`	Size of matte data in bytes	4
`MatteRect`	Rectangle for matte data	8

The `MatteRect` field of opcode $8201 is followed by three variable fields and a subopcode:

- The matte image description, which contains the image description structure for the matte. The size of this variable is specified in the first long integer in this opcode. This field is not included if the `MatteSize` field is 0.
- The matte data, which contains information for the matte. The size of this variable is defined by the `MatteSize` field.
- A subopcode (2 bytes in length) which describes the image and mask and is entirely within the other opcode. Its size is included in the size for the main opcode; hence it is not included if the QuickTime opcode is skipped. This subopcode can be either $98, $99, $9A, or $9B.
- The data for the subopcode variable which contains information for the image.

Compressing Images

The Image Compression Manager provides a rich set of functions that allow applications to compress images. Some of these functions present a straightforward interface that is suitable for applications that need little control over the compression operation. Others permit applications to control the parameters that govern the compression operation.

This section describes the basic steps that your application follows when compressing a single frame of image data. Following this discussion, Listing 3-1 shows a sample function that compresses an image.

First, determine the parameters for the compression operation. Typically, the user specifies these parameters in a user dialog box you may supply via the standard compression dialog component. For comprehensive details, see the chapter “Standard Image-Compression Dialog Components” in *Inside Macintosh: QuickTime Components*. Your application may choose to give the user the ability to specify such parameters as the compression algorithm, image quality, and so on.

Your application may give the user the option to specify a compression algorithm based on an important performance characteristic. For example, the user may be most concerned with size, speed, or quality. The Image Compression Manager allows your application to choose the compressor component that meets the specified criterion.

To determine the maximum size of the resulting compressed image, your application should then call the Image Compression Manager's `GetMaxCompressionSize` function (described on page 3-68). You provide the specified compression parameters to this function. In response, the Image Compression Manager invokes the appropriate compressor component to determine the maximum number of bytes required to store the compressed image. Your application should then reserve sufficient memory to accommodate the compressed image or use a data-unloading function to spool the compressed data to disk (see "Spooling Compressed Data" beginning on page 3-44 for more information about data-unloading functions).

Once the user has specified the compression parameters and your application has established an appropriate environment for the operation, call the `CompressImage` (or `FCompressImage`) function to compress the image. Use the `CompressImage` function (described on page 3-73) if your application does not need to control all the parameters governing compression. If your application needs access to other compression parameters, use the `FCompressImage` function (described on page 3-75).

The Image Compression Manager manages the compression operation and invokes the appropriate compressor. The manager returns the compressed image and its associated image description structure to your application. Note that the image description structure contains a field indicating the size of the resulting image.

Note
You should use the standard compression dialog component to set up the parameters for compression. See the chapter "Standard Image-Compression Dialog Components" in *Inside Macintosh: QuickTime Components* for details. u

Listing 3-1 Compressing and decompressing an image

```
#include <Types.h>
#include <Traps.h>
#include <Memory.h>
#include <Errors.h>
#include <FixMath.h>
#include "Movies.h"
#include "ImageCompression.h"
#include "StdCompression.h"

#define kMgrChoose 0
PicHandle GetQTCompressedPict (PixMapHandle myPixMap);
```

```
PicHandle GetQTCompressedPict( PixMapHandle myPixMap )
{
   long                    maxCompressedSize = 0;
   Handle                  compressedDataH = nil;
   Ptr                     compressedDataP;
   ImageDescriptionHandle  imageDescH = nil;
   OSErr                   theErr;
   PicHandle               myPic = nil;
   Rect                    bounds = (**myPixMap).bounds;
   CodecType               theCodecType = 'jpeg';
   CodecComponent          theCodec = (CodecComponent)anyCodec;
   CodecQ                  spatialQuality = codecNormalQuality;
   short                   depth = 0;/* let ICM choose depth */


   theErr = GetMaxCompressionSize( myPixMap, &bounds, depth,
                                   spatialQuality, theCodecType,
                                   (CompressorComponent)theCodec,
                                    &maxCompressedSize);
   if ( theErr ) return nil;

   imageDescH = (ImageDescriptionHandle)NewHandle(4);
   compressedDataH = NewHandle(maxCompressedSize);
   if ( compressedDataH != nil && imageDescH != nil  )
   {
      MoveHHi(compressedDataH);
      HLock(compressedDataH);
      compressedDataP = StripAddress(*compressedDataH);

      theErr = CompressImage( myPixMap,
                  &bounds,
                  spatialQuality,
                  theCodecType,
                  imageDescH,
                  compressedDataP);

      if ( theErr == noErr )
      {
         ClipRect(&bounds);
         myPic = OpenPicture(&bounds);
         theErr = DecompressImage( compressedDataP,
                              imageDescH,
                              myPixMap,
```

```
                                &bounds,
                                &bounds,
                                srcCopy,
                                nil );
            ClosePicture();
         }
         if ( theErr
               || GetHandleSize((Handle)myPic) == sizeof(Picture) )
         {
            KillPicture(myPic);
            myPic = nil;
         }
      }
      if (imageDescH) DisposeHandle( (Handle)imageDescH);
      if (compressedDataH) DisposeHandle( compressedDataH);

      return myPic;
}
```

Decompressing Images

"Working With Pictures," which begins on page 3-24, discusses how applications can display compressed images that are stored as pictures by calling the `DrawPicture` function. The Image Compression Manager also provides functions that allow your application to display single-frame compressed images. As with image compression, your application can choose to specify all the parameters that govern the operation, or it can leave many of these choices to the Image Compression Manager.

This section describes the steps your application must follow to decompress an image into a pixel map.

First, your application determines where to display the decompressed image. Your application must specify the destination graphics port to the Image Compression Manager. In addition, you may indicate that only a portion of the source image is to be displayed. You describe the desired portion of the image by specifying a rectangle in the coordinate system of the source image. You can determine the size of the source image by examining the image description structure associated with the image (see "The Image Description Structure" on page 3-49 for more information about image description structures).

Your application may also specify that the image is to be mapped into the destination graphics port. The Image Compression Manager provides two mechanisms for mapping images during decompression. The `DecompressImage` function (described on page 3-78) accepts a second rectangle as a parameter. During decompression the Image Compression Manager maps the desired image to the destination rectangle, scaling the resulting image as appropriate to fit the destination rectangle. The `FDecompressImage` function (described on page 3-79) allows your application to specify a mapping matrix

for the operation. Currently, the Image Compression Manager supports only scaling and translation matrix operations.

Your application can invoke further effects by specifying a mask region or blend matte for the image. Mask regions and mattes control which pixels in the source image are drawn to the destination. **Mask regions** define the part of the source image that is displayed. During decompression the Image Compression Manager displays only those pixels in the source image that correspond to bits in the mask that are set to 1. Mask regions must be defined in the destination coordinate system.

Blend mattes contain several bits per pixel and are defined in the coordinate system of the source image. Mattes provide a mechanism for mixing two images. The Image Compression Manager displays the weighted average of the source and destination based on the corresponding pixel in the matte (this feature is fully functional in System 7 and is approximated in System 6).

Decompress the image by calling the Image Compression Manager's `DecompressImage` or `FDecompressImage` function. Your application must provide an image description structure along with the other parameters governing the operation. Use the `DecompressImage` function for simple decompression operations. If your application needs greater control, use the `FDecompressImage` function. See "Working With Images" which begins on page 3-73, for detailed descriptions of these functions.

The Image Compression Manager manages the decompression operation and invokes the appropriate decompressor component. The manager returns the decompressed image to the location specified by your application.

Compressing Sequences

The Image Compression Manager also provides functions that allow your application to compress and decompress sequences of images, such as might constitute a QuickTime movie. The tools provided by the Image Compression Manager focus on image compression and decompression and on the ordering of the images in a sequence, not on timing considerations. Use the Movie Toolbox to handle all the issues relating to the amount of time each image should be shown on the screen. For information on decompressing image sequences, see the next section, "Decompressing Sequences."

A series of images can be compressed as a sequence if those images share an image description. That is, each image in the sequence must have the same compressor type, pixel depth, color lookup table, and boundary dimensions. To take best advantage of temporal compression, the images should also be related to each other (like frames in a movie), but this relationship is not necessary for them to be grouped as a sequence. If you create a sequence from completely unrelated images, you may not be able to achieve significant temporal compression.

When compressing image sequences, your application must perform several steps in addition to those required for single-frame image compression. This section describes a typical function for compressing an image sequence. Note that much of the setup processing is the same as that performed for single-frame images.

First, determine the parameters for the compression operation. As with single-image compression, the user may specify these parameters in a dialog box you can supply via the standard image-compression dialog component (see the chapter "Standard Image-Compression Dialog Components" in *Inside Macintosh: QuickTime Components* for details). Your application may choose to give the user the ability to specify such parameters as the compression algorithm, image quality, and so on. Note that image sequences require additional parameters, such as temporal quality.

Your application may give the user the option of specifying a compression algorithm based on an important performance characteristic. For example, the user may be most concerned with size, speed, or accuracy. The Image Compression Manager allows your application to choose the compressor component that meets the specified criterion.

Your application signals its intention to compress an image sequence by issuing the Image Compression Manager's `CompressSequenceBegin` function (see page 3-106 for more information about this function). At this time your application specifies many of the parameters that govern the sequence-compression operation. When you set the compression parameters and the `temporalQuality` parameter is not 0, then be sure to set the value of either the `codecFlagUpdatePrevious` or `codecFlagUpdatePreviousComp` flag to 1 in the `flags` parameter of the `CompressSequenceBegin` function.

Once you have started the sequence, you then compress each image in the sequence by performing the following steps:

1. Your application must call the Image Compression Manager's `GetMaxCompressionSize` function to determine the maximum size of the compressed data that will result from the current image (see "Getting Information About Compressed Data" on page 3-67 for more information about this function). You provide the specified compression parameters to this function. In response, the Image Compression Manager invokes the appropriate compressor component to determine the number of bytes required to store the largest compressed image in the sequence. Your application should then reserve sufficient memory to accommodate that compressed image. You can use this returned value until you change the settings of the compression parameters.
2. Your application must call the `CompressSequenceFrame` function to compress the image (see "Working With Sequences" on page 3-106 for more information about this function). It may be necessary or desirable for your application to change one or more of the compression parameters while processing a sequence. The Image Compression Manager provides several functions that allow your application to modify such parameters as the spatial or temporal quality or the data-unloading function. See "Changing Sequence-Compression Parameters" on page 3-120 for more information about these functions.
3. The Image Compression Manager manages the compression operation and invokes the appropriate compressor. The manager returns the compressed image and its associated image description to your application.
4. Your application is then free to store the compressed image with the others in the sequence.

After the entire sequence is compressed, you end the process by calling the `CDSequenceEnd` function (see page 3-119 for more information about this function).

Decompressing Sequences

The Movie Toolbox handles the details of displaying compressed image sequences that are stored in QuickTime movies. (For details, see the chapter "Movie Toolbox" in this book.) However, if you want to work with sequences in your application, the Image Compression Manager provides tools for decompressing image sequences. As with still-image compression, decompressing sequences requires additional effort on the part of your application. In addition, there are some processing considerations that are particular to sequence decompression. This section describes the steps necessary to decompress an image sequence. Then it discusses several points you should consider before decompressing a sequence.

When decompressing an image sequence, your application must first determine where to display the decompressed sequence. Your application must specify the destination graphics port to the Image Compression Manager. In addition, you may indicate that only a portion of the source image is to be displayed. You describe the desired portion of the image by specifying a rectangle in the coordinate system of the source image. You can determine the size of the source image by examining the image description structure associated with the image (see "The Image Description Structure" on page 3-49 for more information about image description structures).

Your application may also specify that the image is to be mapped into the destination graphics port. The `DecompressSequenceBegin` function (described on page 3-113) allows your application to specify a mapping matrix for the operation.

Your application can invoke additional effects by specifying a mask region or blend matte for the image. Mask regions and mattes control which pixels in the source image are drawn to the destination. Mask regions must be defined in the destination coordinate system. During decompression the Image Compression Manager displays only those pixels in the source image that correspond to bits in the mask that are set to 1. Mattes contain several bits per pixel and are defined in the coordinate system of the source image. Mattes provide a mechanism for blending pixels from source images.

Your application signals its intention to decompress an image sequence by issuing the Image Compression Manager's `DecompressSequenceBegin` function (see page 3-113 for more information about this function). At this time your application specifies many of the parameters that govern the sequence-decompression operation. The Image Compression Manager, in turn, allocates system resources that are necessary for the operation.

Once you have started the sequence, you then decompress each image in the sequence. Call the `DecompressSequenceFrame` function to decompress the image (described on page 3-116). It may be necessary or desirable for your application to change one or more of the decompression parameters while processing a sequence. The Image Compression Manager provides several functions that allow your application to modify such parameters as the accuracy, the transformation matrix, or the data-loading function. See

"Changing Sequence-Decompression Parameters" beginning on page 3-129 for more information about these functions.

The Image Compression Manager manages the decompression operation and invokes the appropriate compressor component. The manager returns the decompressed image to the location specified by your application and applies any effects you may have specified.

After the entire sequence is decompressed, you end the process by calling the `CDSequenceEnd` function (described on page 3-119).

Decompressing Still Images From a Sequence

Your application can, of course, decompress individual images from a sequence. When doing so, you must be careful to select only those frames that do not depend on other frames. That is, do not decompress frames from a sequence that has been temporally compressed unless you first decompress all the frames in sequence starting from the preceding key frame (see "Defining Key Frame Rates" on page 3-47 for more information on key frames in image sequences). In general, you should decompress images from sequences as sequences, rather than as individual frames.

Using Screen Buffers and Image Buffers

There are two special buffers associated with decompressing an image sequence: a screen buffer and an image buffer. The Image Compression Manager uses the screen buffer to reduce tearing artifacts that result when an image cannot be decompressed to the screen quickly enough. Tearing manifests itself when your eye sees parts of consecutive images simultaneously. Screen buffers should be the same size and pixel depth as the destination. This provides the fastest screen update speed. The compressor decompresses the image to the screen buffer, performing the time-consuming tasks associated with decompression. When the image is fully decompressed, the compressor quickly copies the image to the screen. Few sequences require the use of a screen buffer. You must determine whether it is appropriate to your application.

The Image Compression Manager uses image buffers when decompressing sequences that have been temporally compressed and therefore contain key frames. Image buffers are especially useful when you want to skip to random frames within a sequence. Random frame access in temporally compressed sequences forces the compressor to decompress all the frames between the nearest preceding key frame and the desired frame. Reconstructing the frame in this manner on the screen can result in jerky sequence display. As an alternative, the compressor can reconstruct the frame in the offscreen image buffer and then copy it to the screen when appropriate. Image buffers are allocated at an appropriate depth and size for the decompressor.

Your application can control the use of the image buffer by the compressor component. For example, you can force the compressor to draw images only to the image buffer, not to the screen. In this manner you can use the image buffer to build up sequences without making the process visible. You can also control when the compressor uses the image

buffer. You may need to do this when your program is decompressing directly to the screen and suddenly is prevented from doing so (for example, when your window becomes hidden).

A Sample Program for Compressing and Decompressing a Sequence of Images

The sample program presented in this section illustrates the processes described in the previous sections. The program has been divided into several functions. Listing 3-2 shows the main program.

Listing 3-2 Compressing and decompressing a sequence of images: The main program

```
WindowPtr   displayWindow;    /* window in which to display
                                  sequence */
Rect        windowRect;       /* rectangle of displayWindow */

main (void)
{
   WindowPtr   displayWindow;
   Rect        windowRect;

   InitGraf (&thePort);
   InitFonts ();
   InitWindows ();
   InitMenus ();
   TEInit ();
   InitDialogs (nil);

   SetRect (&windowRect, 0, 0, 256, 256);
   OffsetRect (&windowRect,/* middle of screen */
      ((qd.screenBits.bounds.right - qd.screenBits.bounds.left) -
            windowRect.right) / 2,
      ((qd.screenBits.bounds.bottom - qd.screenBits.bounds.top) -
            windowRect.bottom) / 2);
   displayWindow = NewCWindow (nil, &windowRect,
                               "\pImage", true, 0,
                               (WindowPtr)-1, true, 0);
   if (displayWindow)
   {
      SetPort (displayWindow);
      SequenceSave ();
```

```
      SequencePlay ();
   }
}
```

A Sample Function for Saving a Sequence of Images to a Disk File

The `SequenceSave` function shown in Listing 3-3 saves a sequence of images to a disk file. This function creates and opens a disk file for the image sequence, calls the `CompressSequence` function to create and compress the image sequence into the file, and then calls the `MakeMyResource` function to save the image description resource in the file, so that the sequence can be played back later. For details on `CompressSequence`, see the next section.

The data for each frame is written to the data fork of the disk file, preceded by a long word that contains the number of bytes of data for that frame. A description of the compressed images in the sequence is stored in a `'SEQU'` resource in the same file with a resource ID of 128 or 129. This description is simply the image description structure maintained by the Image Compression Manager.

The image for each frame of the sequence is drawn into an offscreen graphics world that the `SequenceSave` function creates in the `currWorld` variable. `SequenceSave` calls the `DrawOneFrame` function (described in the next section) to draw each frame's image into the `currWorld` variable. Before any of the frames of the sequence are drawn, the Image Compression Manager is prepared to compress a sequence of images through the `CompressSequence` function.

Listing 3-3 Compressing and decompressing a sequence of images: Saving a sequence to a disk file

```
void SequenceSave (void)
{
   long                    filePos;
   StandardFileReply       fileReply;
   short                   dfRef = 0;
   OSErr                   error;
   ImageDescriptionHandle  description = nil;

   StandardPutFile ("\p", "\pSequence File", &fileReply);
   if (fileReply.sfGood)
   {
      if (! (fileReply.sfReplacing))
      {
         error = FSpCreate (&fileReply.sfFile, 'SEQM', 'SEQU',
                            fileReply.sfScript);
         CheckError (error, "\pFSpCreate");
      }
```

```
      error = FSpOpenDF (&fileReply.sfFile, fsWrPerm, &dfRef);
      CheckError (error, "\pFSpOpenDF");

      error = SetFPos (dfRef, fsFromStart, 0);
      CheckError (error, "\pSetFPos");

      CompressSequence (&dfRef, &description);
      error = GetFPos (dfRef, &filePos);
      CheckError (error, "\pGetFPos");

      error = SetEOF (dfRef, filePos);
      CheckError (error, "\pSetEOF");

      FSClose (dfRef);
      FlushVol (nil, fileReply.sfFile.vRefNum);

      MakeMyResource (fileReply, description);
      if (description != nil)
         DisposeHandle ((Handle) description);
   }
}

void MakeMyResource ( StandardFileReply fileReply,
                      ImageDescriptionHandle description)
{
   OSErr    error;
   short    rfRef;
   Handle   sequResource;

   FSpCreateResFile (&fileReply.sfFile, 'SEQM', 'SEQU',
                     fileReply.sfScript);

   error = ResError();
   if (error != dupFNErr)
      CheckError (error, "\pFSpCreateResFile");

   rfRef = FSpOpenResFile (&fileReply.sfFile, fsRdWrPerm);
   CheckError (ResError (), "\pFSpOpenResFile");

   SetResLoad (false);
   sequResource = Get1Resource ('SEQU', 128);
   if (sequResource)
      RmveResource (sequResource);
```

```
    SetResLoad (true);
    sequResource = (Handle) description;
    error = HandToHand (&sequResource);
    CheckError (error, "\pHandToHand");

    AddResource (sequResource,'SEQU', 128, "\p");
    CheckError (ResError (), "\pAddResource");
    UpdateResFile (rfRef);
    CheckError (ResError (), "\pUpdateResFile");

    CloseResFile (rfRef);
}
```

A Sample Function for Creating, Compressing, and Drawing a Sequence of Images

Listing 3-4 shows the `CompressSequence` function, which creates and then compresses the image sequence. `CompressSequenceBegin` informs the Image Compression Manager which compressor (of type `codectype`) to use, what the desired compression quality is, the key frame rate, the portion of the image to compress (in this example, the entire image is compressed), and the image to be compressed (in this example, the pixel map [of type `PixMap`] in the `currWorld` variable).

`CompressSequenceBegin` returns a unique number that identifies the sequence for subsequent image-compression routines, and it initializes a new image description structure, which is stored in the handle referenced by the `description` local variable.

Using a loop, the `DrawOneFrame` function draws each frame until the last frame is drawn, at which time the function returns the value of `false`. Each frame that it draws is copied to the window so that it can be seen during the compression sequence.

The `CompressSequenceFrame` function is used to compress each frame's image. `CompressSequenceFrame` tells the Image Compression Manager

- which image to compress (in this case, the pixel map of the `currWorld` variable)
- the portion of that image to compress (in this case, all of it)
- whether to update the previous frame's buffer for frame differencing
- the address of the buffer that's to receive the compressed image data

In updating the previous frame's buffer for frame differencing, the Image Compression Manager control flag `codecFlagUpdatePrevious` copies the uncompressed image to the previous frame's buffer; contrast this with the `codecFlagUpdatePreviousComp` flag, which copies the compressed image to the previous frame's buffer—the more lossy the compression, the more the difference between the compressed and uncompressed images.

The `CompressSequenceBegin` function returns a rating of the similarity between the current frame and the previous frame, but this example ignores this rating. After each frame is compressed, the number of bytes in the compressed image data is written to the disk file, followed by the compressed image data itself.

After all the images in the sequence have been compressed, the `CDSequenceEnd` function is called to tell the Image Compression Manager that the sequence is over. The data fork of the file is closed, and the image description is written to a `'SEQU'` resource.

The `DrawOneFrame` function draws one frame of the sequence with QuickDraw. The frame's image is drawn into the rectangle specified by the `destRect` parameter. The image is a set of **color ramps** in which the shading goes from light to dark in smooth increments. The color ramps fill the destination rectangle and the current frame number centered within the destination rectangle over the ramps.

The `PaintImage` function paints a series of vertical color ramps into the rectangle specified by the `destRect` parameter into the current color graphics port. This is done through a nested loop. The outer loop iterates only twice, and half of the ramps are drawn in the first iteration and half in the second. The inner loop iterates over all the steps in a ramp.

Listing 3-4 Compressing and decompressing a sequence of images: Drawing one frame with QuickDraw

```
void CompressSequence (short* dfRef,
                           ImageDescriptionHandle* description)
{
   GWorldPtr      currWorld = nil;
   PixMapHandle   currPixMap;
   CGrafPtr       savedPort;
   GDHandle       savedDevice;
   Handle         buffer = nil;
   Ptr            bufferAddr;
   long           compressedSize;
   long           dataLen;
   Rect           imageRect;
   ImageSequence  sequenceID = 0;
   short          frameNum;
   OSErr          error;
   CodecType      codecKind = 'rle ';

   GetGWorld (&savedPort, &savedDevice);
   imageRect = savedPort->portRect;
   error = NewGWorld (&currWorld, 32, &imageRect, nil, nil, 0);
   CheckError (error, "\pNewGWorld");
   SetGWorld (currWorld, nil);
```

```
   currPixMap = currWorld->portPixMap;
   LockPixels (currPixMap);

/*
   Allocate an embryonic image description structure and the
   Image Compression Manager will resize.
*/
   *description = (ImageDescriptionHandle) NewHandle (4);

   error = CompressSequenceBegin (
         &sequenceID,
         currPixMap,
         nil,                /* tell ICM to allocate previous
                                image buffer */
         &imageRect,
         &imageRect,
         0,                  /* let ICM choose pixel depth */
         codecKind,
         (CompressorComponent) anyCodec,
         codecNormalQuality,  /* spatial quality */
         codecNormalQuality,  /* temporal quality */
         5,                  /* at least 1 key frame every
                                5 frames */
         nil,                /* use default color table */
         codecFlagUpdatePrevious,
         *description );
   CheckError (error, "\pCompressSequenceBegin");

   error = GetMaxCompressionSize(
         currPixMap,
         &imageRect,
         0,                     /* let ICM choose pixel depth */
         codecNormalQuality,    /* spatial quality */
         codecKind,
         (CompressorComponent) anyCodec,
         &compressedSize );
   CheckError (error, "\pGetMaxCompressionSize");

   buffer = NewHandle(compressedSize);
   CheckError (MemError(), "\pNewHandle buffer");
   MoveHHi (buffer);
   HLock (buffer);
   bufferAddr = StripAddress (*buffer);
```

```
   for (frameNum = 1; frameNum <= 10; frameNum++)
   {
      DrawFrame (&imageRect, frameNum);

      error = CompressSequenceFrame (
                  sequenceID,
                  currPixMap,
                  &imageRect,
                  codecFlagUpdatePrevious,
                  bufferAddr,
                  &compressedSize,
                  nil,
                  nil );
      CheckError (error, "\pCompressSequenceFrame");

      dataLen = 4;
      error = FSWrite (*dfRef, &dataLen, &compressedSize);
      CheckError (error, "\pFSWrite length");

      error = FSWrite (*dfRef, &compressedSize, bufferAddr);
      CheckError (error, "\pFSWrite buffer");
   }

   CDSequenceEnd (sequenceID);

   DisposeGWorld (currWorld);
   SetGWorld (savedPort,savedDevice);
   if (buffer) DisposeHandle ( buffer );
   }

void DrawFrame (const Rect *imageRect,  long frameNum)
{
   Str255 numStr;

   ForeColor( redColor );
   PaintRect( imageRect );

   ForeColor( blueColor );
   NumToString (frameNum, numStr);
   MoveTo ( imageRect->right / 2, imageRect->bottom / 2);
   TextSize ( imageRect->bottom / 3);
   DrawString (numStr);
}
```

A Sample Function for Decompressing and Playing Back a Sequence From a Disk File

The `SequencePlay` function, shown in Listing 3-5, plays back a sequence of images from a disk file that was created by the `SequenceSave` function (see Listing 3-3 on page 3-36 for details).

The `SequencePlay` function begins by grabbing the image description structure from the file that the user specified from a `'SEQU'` resource ID 128. This structure is needed to decompress the images in the file.

Before these compressed images are read, the Image Compression Manager is told to prepare to decompress a sequence of images through the `DecompressSequenceBegin` function. This routine tells the Image Compression Manager

- how the images were compressed with the image description structure
- where to display the decompressed image (the current port in this example)
- what part of the image to decompress (all of it)
- what transfer mode to use when displaying the image (`srcCopy`)
- whether to buffer the image for frame differences

A loop iterates for each frame in the file. For each frame, a long word with the number of bytes in the frame is read from the file, and then that many bytes are read from the file into a compressed-image buffer. This buffer is passed to `DecompressSequenceFrame`, which decompresses the image to the screen (the destination doesn't have to be the screen, but it is in this example). The loop iterates until the end of the file has been reached.

Listing 3-5 Compressing and decompressing a sequence of images: Decompressing and playing back a sequence from a disk file

```
void SequencePlay (void)
{
   ImageDescriptionHandle  description;
   long                    compressedSize;
   Handle                  buffer = nil;
   Ptr                     bufferAddr;
   long                    dataLen;
   long                    lastTicks;
   ImageSequence           sequenceID;
   Rect                    imageRect;
   StandardFileReply       fileReply;
   SFTypeList              typeList = {'SEQU',0,0,0};
   short                   dfRef = 0;     /* sequence data fork */
   short                   rfRef = 0;     /* sequence resource fork */
   OSErr                   error;
```

```
StandardGetFile (nil, 1, typeList, &fileReply);
if (!fileReply.sfGood) return;

rfRef = FSpOpenResFile (&fileReply.sfFile, fsRdPerm);
CheckError (ResError (), "\pFSpOpenResFile");

description = (ImageDescriptionHandle)
                    Get1Resource ('SEQU', 128);
CheckError (ResError (), "\pGet1Resource");

DetachResource ((Handle) description );
HNoPurge ((Handle) description );
CloseResFile (rfRef);

error = FSpOpenDF (&fileReply.sfFile, fsRdPerm, &dfRef);
CheckError (error, "\pFSpOpenDF");

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

SetRect (&imageRect, 0, 0, (**description).width,
                           (**description).height);
error = DecompressSequenceBegin (
               &sequenceID,
               description,
               nil,                /* use the current port */
               nil,                /* go to screen */
               &imageRect,
               nil,                /* no matrix */
               ditherCopy,
               nil,                /* no mask region */
               codecFlagUseImageBuffer,
               codecNormalQuality,  /* accuracy */
               (CompressorComponent) anyCodec);

while (true)
{
   dataLen = 4;
   error = FSRead (dfRef, &dataLen, &compressedSize);
   if (error == eofErr)
      break;
   CheckError( error, "\pFSRead" );
```

```
      if (compressedSize > GetHandleSize (buffer))
      {
         HUnlock (buffer);
         SetHandleSize (buffer, compressedSize);
         CheckError (MemError(), "\pSetHandleSize");
      }

      HLock (buffer);
      bufferAddr = StripAddress (*buffer);
      error = FSRead (dfRef, &compressedSize, bufferAddr);
      CheckError (error, "\pFSRead");

      error = DecompressSequenceFrame (
                        sequenceID,
                        bufferAddr,
                        0, // flags
                        nil,
                        nil );
      CheckError (error, "\pDecompressSequenceFrame");

      Delay (30, &lastTicks);
   }

   CDSequenceEnd (sequenceID);
   if (dfRef) FSClose (dfRef);
   if (buffer) DisposeHandle (buffer);
   if (description) DisposeHandle ((Handle)description);
}
```

Spooling Compressed Data

During compression and decompression operations it may be necessary to spool the image data to or from storage other than computer memory. If your application uses the Image Compression Manager functions that handle picture files, the Image Compression Manager manages this spooling for you. However, if you use the functions that work with pixel maps or sequences and your application cannot store the image data in memory, it is your application's responsibility to spool the data.

The Image Compression Manager provides a mechanism that allows the compressors and decompressors to invoke spooling functions provided by your application. There are two kinds of data-spooling functions: data-loading functions and data-unloading functions. Decompressors call data-loading functions during image decompression. The data-loading function is responsible for providing compressed image data to the

decompressor. The decompressor then decompresses the data and writes the resulting image to the appropriate location. See “Application-Defined Functions” beginning on page 3-148 for a detailed description of the calling sequence used by the decompressor component when it invokes your data-loading function.

Compressors call data-unloading functions during image compression. The data-unloading function must remove the compressed image data from memory. The compressor can then compress more of the image and write the compressed image data into the available buffer space. See “Application-Defined Functions” beginning on page 3-148 for a detailed description of the calling sequence used by the compressor component when it invokes your data-unloading function.

When compressing sequences, your application assigns a data-unloading function by calling the `SetCSequenceFlushProc` function (described on page 3-125). When decompressing sequences, you assign a data-loading function by calling the `SetDSequenceDataProc` function (described on page 3-135).

When your application assigns a spooling function to an image or sequence operation, you must also specify a data buffer and the size of that buffer. The `codecMinimumDataSize` value specifies the smallest data buffer you may allocate for image data spooling.

```
#define codecMinimumDataSize 32768     /* minimum data size */
```

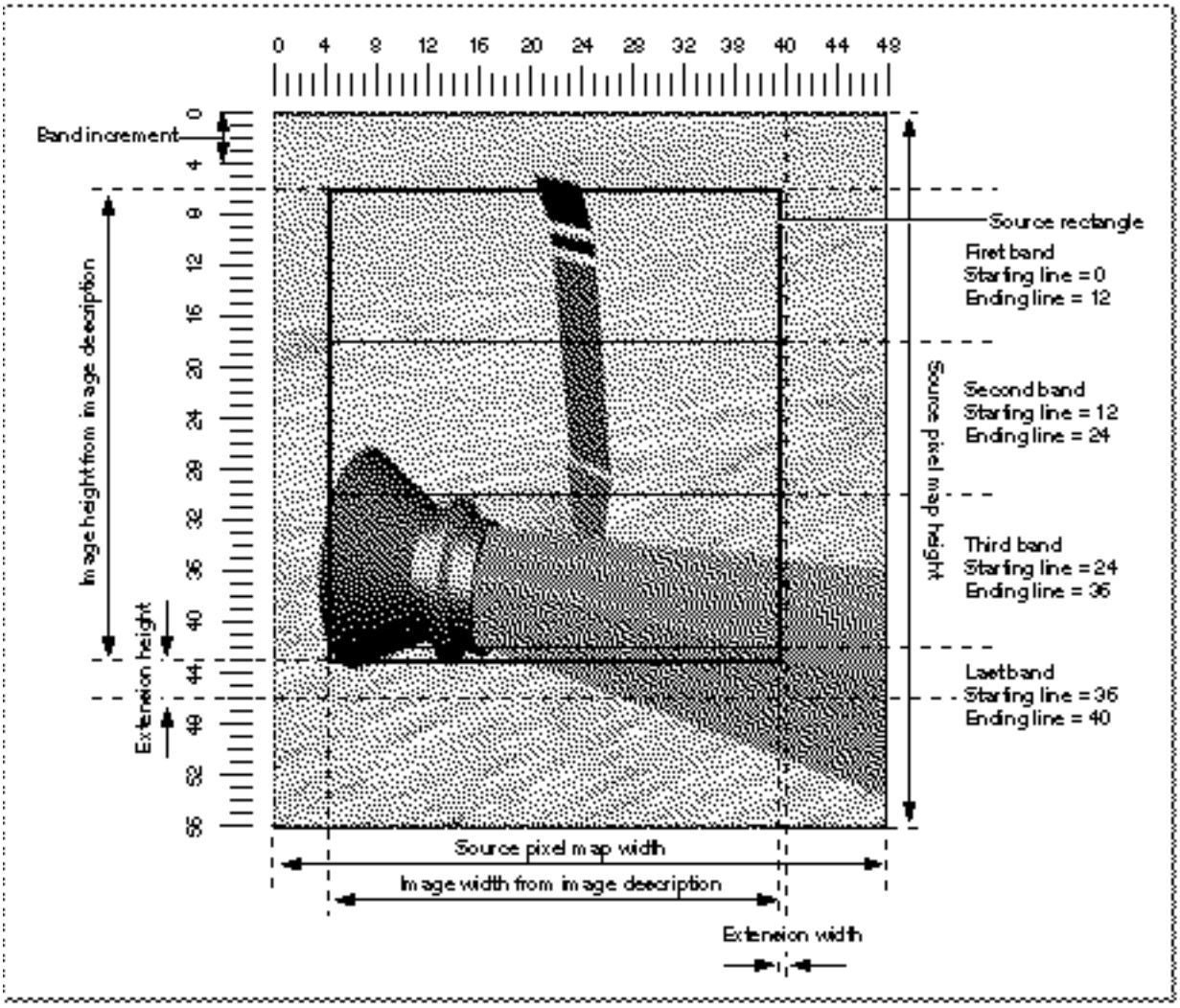
Banding and Extending Images

Occasionally a compressor component may not be able to accommodate the destination rectangle for an image decompression or the source for an image-compression operation. This situation may result from compressors that are optimized to work at certain depths or that cannot perform scaling, translation, dithering, or masking during decompression. In such circumstances the Image Compression Manager allocates a temporary buffer that is acceptable to the compressor component and breaks the image up to fit into that new buffer. Since there often is not enough memory to allocate a buffer to hold the entire image, the Image Compression Manager may allocate one that holds a band of the image. A **band** is one horizontal piece of the image. Its height is some portion of the desired image height (before scaling or rotation), and it is at least as wide as the desired image.

The height of the band is determined both by the amount of memory available and the block size of the compressor component. The block size of a compressor is the natural size at which it handles images, and it is peculiar to the image-compression algorithm. The block size for the photo compressor is usually 16 pixels by 16 pixels, for example. Usually the block width and height are equal, but this is not always the case. The minimum height of a band is one strip of blocks. A strip is defined to be a part of an image that is as high as the block height (for the compressor in question) and as wide as the band. The width of a band is either the width of the desired unscaled image, or that width increased by an extension.

Figure 3-9 shows the measurements of several image bands.

Figure 3-9 Image bands and their measurements



Some compressors can only handle images with dimensions that are a multiple of their block size. If the desired image does not comply with this restriction in either dimension, the Image Compression Manager extends the band on the right side and bottom by the amount required to meet the needs of the compressor. During compression, the compressor fills the extended region with the same pixel value as the pixels adjacent to the extension. During decompression, the Image Compression Manager writes only the pixels that are part of the source image. The extended portion remains only in the offscreen buffer.

Defining Key Frame Rates

The process of temporal compression involves reducing or eliminating temporal redundancy from an image sequence. Temporal compression is most effective when a sequence contains frames that bear significant similarity to adjacent frames. This is typically true of movies and other video sequences. Reconstructing an individual frame within a sequence that has been temporally compressed requires knowledge of the previous frames. This does not present a problem if your application always plays compressed sequences from the beginning. However, if your application needs to start playing a sequence from a random point, or perhaps backward, the decompressor does not have enough information to decompress the frames.

To alleviate this problem, compressors insert key frames in compressed sequences at regular intervals. **Key frames** define starting points for portions of a temporally compressed sequence. Subsequent frames depend on the previous key frame.

At the start of a sequence compression your application can specify a rate at which the compressor is to insert key frames into the compressed data stream. This **key frame rate** indicates the maximum number of frames you will accept between key frames. The Image Compression Manager picks the best key frames from the source sequence and at the same time enforces the specified key frame rate (the best key frames are those that are least similar to adjacent frames, such as at scene changes—these frames would have the largest compressed images even if they were not selected as key frames).

During sequence compression your application can change the key frame rate by calling the `SetCSequenceKeyFrameRate` function (described beginning on page 3-121). By manipulating the parameters for the sequence, you can force the Image Compression Manager to place a key frame at any arbitrary point in a sequence (set the `codecFlagForceKeyFrame` flag to 1 in the `flags` parameter of the `CompressSequenceFrame` function—described beginning on page 3-111).

Fast Dithering

QuickDraw provides a means of displaying images with high color resolution in pixel maps or on screens with lower color resolution. By dithering the destination image, QuickDraw fools your eyes into seeing colors that are not actually available on the display screen. Unfortunately, the error-diffusion technique used by QuickDraw takes longer than just drawing pixels by directly looking them up in a color table. The drawing delays imposed by standard dithering are unacceptable when working with movies.

To alleviate this problem, Apple has developed a technique that allows faster dithering to destinations that use 8 bits per pixel. Fast dithering uses lookup tables created by the Image Compression Manager. All the decompressors supplied by Apple can use fast dithering.

Apple decompressors use fast dithering when copying from image band buffers to 8-bit destinations. If the accuracy for decompression is above normal, then the decompressors use true error diffusion rather than fast dithering. Note that video sequences are normally displayed at normal or low accuracy so that you can obtain maximum display speed during decompression.

Understanding Compressor Components

This section discusses key attributes of compressor components and the functional interfaces these components must support. (**Compressor components** here refers to both image compressor components and image decompressor components.) This information is intended for developers of compressor components. Application developers do not need to be familiar with this material to use the Image Compression Manager.

A compressor component is a code resource that provides image compression or decompression services for image data. These components may also utilize additional hardware to provide their services. Compressor components are registered by the Component Manager, and they present a standard set of function interfaces to the Image Compression Manager (see *Inside Macintosh: QuickTime Components* for a detailed description of the functions that compressors must provide). A compressor can be a systemwide resource, or it can be local to a particular application.

Applications never communicate directly with compressors. Applications request compressor services by issuing the appropriate Image Compression Manager functions. The Image Compression Manager then performs its necessary processing before invoking the compressor. Of course, an application could install its own compressor component. However, any interaction between the application and the compressor is still managed by the Image Compression Manager.

The Image Compression Manager knows about two types of compressor components. Components that can compress image data carry a component type (described by the `compressorComponentType` data type) of `'imco'` and are referred to as *compressors*. Components that can decompress images have a component type (described by the `decompressorComponentType` data type) of `'imdc'` and are called *decompressors*. The value of the component subtype indicates the compression algorithm supported by the component. All compressor components with the same subtype must be able to handle the same format of compressed data. During decompression a component should handle all variations of the data specified for a subtype. Conversely, while compressing an image a compressor must not produce data that decompressors of the same subtype cannot handle during decompression.

The Image Compression Manager defines four callback functions that may be provided to compressors or decompressors by applications. A **callback function** is an application-defined function that is invoked at a specified time or based on specified criteria. These callback functions are data-loading functions, data-unloading functions, completion functions, and progress functions. Data-loading functions and data-unloading functions support spooling of compressed data. Completion functions allow compressors and decompressors to report that asynchronous operations have completed. Progress functions provide a mechanism for compressors and decompressors to report their progress toward completing an operation. For more information about these callback functions, see “Application-Defined Functions” beginning on page 3-148.

Image Compression Manager Reference

This section describes all of the Image Compression Manager functions and data structures. The Image Compression Manager provides a rich and varied set of functions that allow your application to work with compressed image data. This discussion has been divided into the following sections:

- "Data Types" identifies the data structures used by your application when interacting with the Image Compression Manager.
- "Image Compression Manager Functions" describes the functions that your application can use to work with compressed data.
- "Application-Defined Functions" describes the interfaces to the callback functions that may be provided to compressors or decompressors by applications.

Data Types

This section describes the format and content of the data structures, data types, and constants that you use to exchange information with the Image Compression Manager.

The Image Description Structure

An image description structure contains information that defines the characteristics of a compressed image or sequence. Data in the image description structure indicates the type of compression that was used, the size of the image when displayed, the resolution at which the image was captured, and so on. One image description structure may be associated with one or more compressed frames.

The `ImageDescription` data type defines the layout of an image description structure. In addition, an image description structure may contain additional data in extensions and custom color tables. The Image Compression Manager provides functions that allow you to get and set the data in image description structure extensions and custom color tables.

- See "Working With Images," which begins on page 3-73, for more information about the functions `GetImageDescriptionCTable` and `SetImageDescriptionCTable`, which allow you to work with custom color tables in image description structures.
- See *Inside Macintosh: QuickTime Components* for more information about the `GetImageDescriptionExtension`, `SetImageDescriptionExtension`, `RemoveImageDescriptionExtension`, `CountImageDescriptionExtensionType`, and `GetNextImageDescriptionExtensionType` functions, which allow you to work with image description structure extensions.

```
struct ImageDescription {
   long idSize;      /* total size of this structure */
   CodecType cType;  /* compressor creator type */
   long resvd1;      /* reserved--must be set to 0 */
   short resvd2;     /* reserved--must be set to 0 */
   short dataRefIndex;/* reserved--must be set to 0 */
   short version;    /* version of compressed data */
   short revisionLevel;/* compressor that created data */
   long vendor;      /* compressor developer that created data */
   CodecQ temporalQuality;
                     /* degree of temporal compression */
   CodecQ spatialQuality;
                     /* degree of spatial compression */
   short width;      /* width of source image in pixels */
   short height;     /* height of source image in pixels */
   Fixed hRes;       /* horizontal resolution of source image */
   Fixed vRes;       /* vertical resolution of source image */
   long dataSize;    /* size in bytes of compressed data */
   short frameCount; /* number of frames in image data */
   Str31 name;       /* name of compression algorithm */
   short depth;      /* pixel depth of source image */
   short clutID;     /* ID number of the color table for image */
};
typedef struct ImageDescription ImageDescription;
typedef ImageDescription *ImageDescriptionPtr,
**ImageDescriptionHandle;
```

Field descriptions

`idSize` Defines the total size of this image description structure with extra data including color lookup tables and other per sequence data.

`cType` Indicates the type of compressor component that created this compressed image data. The value of this field indicates the compression algorithm supported by the component. The `Codec` data type defines a field in the compressor name list structure that identifies the compression method employed by a given compressor component. Apple Computer's Developer Technical Support group assigns these values so that they remain unique. These values correspond, in turn, to text strings that can identify the compression method to the user. See the description of `GetCodecNameList` on page 3-63 for a list of valid values.

`resvd1` Reserved for Apple. This field must be set to 0.

`resvd2` Reserved for Apple. This field must be set to 0.

`dataRefIndex` Reserved for Apple. This field must be set to 0.

`version` Indicates the version of the compressed data. The contents of this field should indicate the version of the compression algorithm that

was used to create the compressed data. By examining this field, decompressors that support many versions of an algorithm can determine the proper way to decompress the image.

`revisionLevel` Indicates the version of the compressor that created the compressed image. Developers of compressors and decompressors assign these version numbers.

`vendor` Identifies the developer of the compressor that created the compressed image.

`temporalQuality`
Indicates the degree of temporal compression performed on the image data associated with this description. This field is valid only for sequences. See "Compression Quality Constants" beginning on page 3-57 for a list of available values.

`spatialQuality`
Indicates the degree of spatial compression performed on the image data associated with this description. This field is valid for sequences and still images. See "Compression Quality Constants" on page 3-57 for a list of available values.

`width` Contains the width of the source image, in pixels.

`height` Contains the height of the source image, in pixels.

`hRes` Contains the horizontal resolution of the source image, in dots per inch.

`vRes` Contains the vertical resolution of the source image, in dots per inch.

`dataSize` Indicates the size of the compressed image, in bytes. This field is valid only for still images. Set this field to 0 if the size is unknown.

`frameCount` Contains the number of frames in the image data associated with this description.

`name` Indicates the compression algorithm used to create the compressed data. This algorithm is stored in Pascal string format. It always takes up 32 bytes no matter how long the string is. The 32 bytes consist of 31 bytes plus one length byte. The value of this field should correspond to the compressor type specified by the `cType` field, as well as to the value of the `typeName` field in the appropriate compressor name structure returned by the `GetCodecNameList` function (see "The Compressor Name List Structure" on page 3-56 for information on the compressor list name structure; see "Getting Information About Compressor Components," which begins on page 3-62, for information on the `GetCodecNameList` function). Applications may use the contents of this field to indicate the type of compression used for the associated image.

`depth` Contains the pixel depth specified for the compressed image. Values of 1, 2, 4, 8, 16, 24, and 32 indicate the depth of color images. Values of 34, 36, and 40 indicate 2-bit, 4-bit, and 8-bit grayscale, respectively, for grayscale images.

`clutID` Contains the ID of the color table for the compressed image, or other special values. If this field is set to 0, then a custom color table is defined for the compressed image. You can use the `GetImageDescriptionCTable` function, described on page 3-87, to retrieve the color table. If this field is set to –1, the image does not use a color table.

The Compressor Information Structure

Your application can retrieve information describing the capabilities of compressors with the `GetCodecInfo` function (described on page 3-65). The `CodecInfo` data type defines the format of the compressor information structure.

```
/* compressor information structure */
struct CodecInfo {
   Str31 typeName;      /* compression algorithm (codec type) */
   short version;       /* version supported by component */
   short revisionLevel; /* version assigned by developer */
   long vendor;         /* developer of component */
   long decompressFlags; /* decompression capability flags */
   long compressFlags;  /* compression capability flags */
   long formatFlags;    /* compression format flags */
   unsigned char compressionAccuracy;
                        /* relative accuracy of this algorithm */
   unsigned char decompressionAccuracy;
                        /* relative accuracy of this algorithm */
   unsigned short compressionSpeed;
                        /* relative compression speed */
   unsigned short decompressionSpeed;
                        /* relative decompression speed */
   unsigned char compressionLevel;
                        /* relative compression of component */
   char resvd;          /* reserved--set to 0 */
   short minimumHeight; /* minimum image height for component */
   short minimumWidth;  /* minimum image width for component */
   short decompressPipelineLatency;
                        /* in milliseconds (asynchronous) */
   short compressPipelineLatency;
                     /* in milliseconds (asynchronous) */
   long privateData; /* reserved for use by Apple */
};
typedef struct CodecInfo CodecInfo;
```

Field descriptions

`typeName` Indicates the compression algorithm used by the component—for example, `'Animation'`. This Pascal string may be used to identify the compression algorithm to the user. The string always takes up 32 bytes no matter how long it is. The 32 bytes consist of 31 bytes plus one length byte. Apple Computer's Developer Technical Support group assigns these type names. The value of this field should correspond to the value of the `typeName` field in the appropriate compressor name structure returned by the `GetCodecNameList` function (see "The Compressor Name Structure" on page 3-55 for information on the compressor name structure; see page 3-63 for information on the `GetCodecNameList` function).

`version` Indicates the version of compressed data this component supports. The contents of this field should indicate the most recent version of the compression algorithm that the component can understand.

`revisionLevel` Indicates the version of the component—for example, `0x00010001 (1.0.1)`. Developers of compressors assign these version numbers.

`vendor` Identifies the developer of the component—for example, `'appl'`. The value of this field corresponds to the manufacturer code or application signature assigned to the developer.

`decompressFlags` Contains flags that specify the decompression capabilities of the component. Typically, these flags are of interest only to developers of image decompressors. The bit values for this field are described in the discussion of image decompressors in *Inside Macintosh: QuickTime Components*.

`compressFlags` Contains flags that specify the compression capabilities of the component. Typically, these flags are of interest only to developers of image compressors. The bit values for this field are described in the discussion of image compressors in *Inside Macintosh: QuickTime Components*.

`formatFlags` Contains flags that describe the possible format for compressed data produced by this component and the format of compressed files that the component can handle during decompression. Typically, these flags are of interest only to developers of compressor components. The bit values for this field are described in the discussion of image compressor and decompressor components in *Inside Macintosh: QuickTime Components*.

`compressionAccuracy` Indicates the relative accuracy of the compression algorithm employed by the component. Valid values for this field range from 0 to 255. A value of 0 means that the accuracy is unknown. Values from 1 to 255 provide a gauge for the relative accuracy of the compression algorithm—higher values indicate better accuracy.

The Image Compression Manager examines this field to determine which compressor component can most accurately compress a given image.

The `compressionAccuracy` field can only approximate the accuracy of a compression algorithm. Typically, compression algorithms produce results of varying quality based on a variety of parameters, including image size and content. Since this information is not available until a compression request is issued, a precise measure of accuracy is not possible. However, the value of this field should still give a rough idea of the accuracy of the supported algorithm.

`decompressionAccuracy`

Indicates the relative accuracy of the decompression algorithm employed by the component. Valid values for this field range from 0 to 255. A value of 0 means that the accuracy is unknown. Values from 1 to 255 indicate the relative accuracy of the decompression technique—higher values mean better accuracy.

The Image Compression Manager examines this field to determine which decompressor component can most accurately decompress a given image.

The `decompressionAccuracy` field can only approximate the accuracy of a decompression algorithm. Typically, decompression algorithms produce results of varying quality based on a variety of parameters, including image size and content. Since this information is not available until a decompression request is issued, a precise measure of accuracy is not possible. However, the value of this field should still give a rough idea of the accuracy of the supported algorithm.

`compressionSpeed`

Indicates the relative speed of the component for compression operations. Valid values for this field lie in the range from 0 to 65,535. A value of 0 means that the speed is unknown. Values from 1 to 65,535 correspond to the number of milliseconds the component requires to compress a 320-by-240 pixel image on a Macintosh II computer.

The Image Compression Manager examines this field to determine which compressor component can most quickly compress a given image.

`decompressionSpeed`

Indicates the relative speed of the component for decompression operations. Valid values for this field lie in the range from 0 to 65,535. A value of 0 means that the speed is unknown. Values from 1 to 65,535 correspond to the number of milliseconds the component requires to decompress a 320-by-240 pixel image on a Macintosh II computer.

The Image Compression Manager examines this field to determine which compressor component can most quickly decompress a given image.

`compressionLevel`
: Indicates the relative compression achieved by this component. Valid values for this field lie in the range from 0 to 255. A value of 0 means that the compression level is unknown. Values from 1 to 255 map to percentage values of relative compression—lower values mean lesser compression. A value of 1 means no compression (0 percent); a value of 255 means maximum compression (100 percent).

  The Image Compression Manager examines this field to determine which available compressor component will yield the smallest resulting data for a given image.

  The `compressionLevel` field can only approximate the effectiveness of a compression algorithm. Typically, compression algorithms produce results of varying quality based on a variety of parameters, including image size and content. Since this information is not available until a compression request is issued, a precise measure of compression is not possible. However, the value of this field should still give a rough idea of the effectiveness of the supported algorithm.

`resvd`
: Reserved for Apple. This field must be set to 0.

`minimumHeight`
: Specifies the height in pixels of the smallest image the component can handle. Together with the `minimumWidth` field, this field defines the block size for the component. The Image Compression Manager does not issue compression or decompression requests for images smaller than the block size.

`minimumWidth`
: Specifies the width in pixels of the smallest image the component can handle. Together with the `minimumHeight` field, this field defines the block size for the component. The Image Compression Manager does not issue compression or decompression requests for images smaller than the block size.

`decompressPipelineLatency`
: Reserved for future use. This field must be set to 0.

`compressPipelineLatency`
: Reserved for future use. This field must be set to 0.

`privateData`
: Reserved for use by Apple. This field must be set to 0.

The Compressor Name Structure

The `CodecNameSpec` data type defines a compressor name structure.

```
/* compressor name structure from GetCodecNameList function */
struct CodecNameSpec
{
   CodecComponent codec;/* component ID for compressor */
   CodecType cType;     /* type identifier for compressor */
   Str31 typeName;      /* string identifier of algorithm */
```

```
   Handle name;         /* name of compressor component */
};
typedef struct CodecNameSpec CodecNameSpec;
```

Field descriptions

`codec` Uniquely identifies the component or, in some cases, contains a special value that selects all components. If your application requests a list of components, the `codec` field in each compressor name structure contains the component ID for that compressor. If your application requests a list of component types, the `codec` field is set to 0 in each compressor name structure.

`cType` Contains the type identifier for the compressor. The value of this field indicates the compression algorithm supported by the component. See the description of `GetCodecNameList` on page 3-63 for a list of valid values.

`typeName` Contains a text string in Pascal format that identifies the compression algorithm supported by the component. This string may be used to identify the compression algorithm to the user. The value of this field should correspond to the value of the `typeName` field in the appropriate compressor information structure returned by the component in response to a `GetCodecInfo` function (see "The Compressor Information Structure" on page 3-52 for information on the compressor information structure; see page 3-65 for information on the `GetCodecInfo` function).

`name` Specifies the name of the compressor component. Developers assign these names to uniquely identify their products. This name may be used to identify the component to the user.

The Compressor Name List Structure

The compressor name list structure contains a list of compressor name structures. (A compressor name structure identifies a compressor or decompressor component.) The data structure contains name and type information for the component. The `GetCodecNameList` function returns an array of these structures, formatted into a compressor name list structure. See page 3-63 for more information on the `GetCodecNameList` function. The `CodecNameSpecList` data type defines a compressor name list structure.

```
/* compressor name list structure */
struct CodecNameSpecList {
   short count;   /* how many compressor name structures */
   CodecNameSpec list[1];
                  /* array of compressor name structures */
};
typedef struct CodecNameSpecList CodecNameSpecList;
typedef CodecNameSpecList *CodecNameSpecListPtr;
```

Field descriptions

`count`	Indicates the number of compressor name structures contained in the `list` array that follows.
`list`	Contains an array of compressor name structures. Each structure corresponds to one compressor component or type that meets the selection criteria your application specifies when it issues the `GetCodecNameList` function. The `count` field indicates the number of structures stored in this array.

Compression Quality Constants

Compressor components may allow applications to assert some control over the image quality that results from a compression or decompression operation. For example, the `CompressSequenceBegin` function (described on page 3-106) provides the `spatialQuality` and `temporalQuality` parameters so that applications can indicate the level of image accuracy desired within individual frames and across adjacent frames in a sequence, respectively. These quality values become a property of the compressed data and are stored in the image description structure (described on page 3-49) associated with the image or sequence.

For a given compression operation, your application can determine the quality that the component supports by issuing the `GetCompressionTime` function (described on page 3-69).

The `CodecQ` data type defines a field that identifies the quality characteristics of a given image or sequence. Note that individual components may not implement all the quality levels shown here. In addition, components may implement other quality levels in the range from `codecMinQuality` to `codecMaxQuality`. Relative quality should scale within the defined value range. Values above `codecLosslessQuality` are reserved for use by individual components.

```
/* compression quality values */
#define codecMinQuality 0x000L   /* minimum valid value */
#define codecLowQuality 0x100L   /* low-quality reproduction */
#define codecNormalQuality
                        0x200L   /* normal-quality repro */
#define codecHighQuality
                        0x300L   /* high-quality repro */
#define codecMaxQuality 0x3FFL   /* maximum-quality repro */
#define codecLosslessQuality
                        0x400L   /* lossless-quality repro */
typedef unsigned long CodecQ;
```

Constant descriptions

`codecMinQuality`
: Specifies the minimum valid value for a `CodecQ` field.

`codecLowQuality`
: Specifies low-quality image reproduction. This value should correspond to the lowest image quality that still results in acceptable display characteristics.

`codecNormalQuality`
: Specifies image reproduction of normal quality.

`codecHighQuality`
: Specifies high-quality image reproduction. This value should correspond to the highest image quality that can be achieved with reasonable performance.

`codecMaxQuality`
: Specifies the maximum standard value for a `CodecQ` field.

`codecLosslessQuality`
: Specifies lossless compression or decompression. This special value is valid only for components that can support lossless compression or decompression.

Image Compression Manager Function Control Flags

A number of Image Compression Manager functions take control flags that allow your application to exert greater control over the operation. In some cases, the Image Compression Manager returns status information about the results of the function in the same flags field. In general, you need to use only a few of these flags. The function descriptions in the reference section of this chapter indicate the flags that are valid for individual functions.

The `CodecFlags` data type defines these flag fields.

```
typedef unsigned short CodecFlags;

/* Image Compression Manager function control flags */
#define codecFlagUseImageBuffer (1L<<0)
                                /* (input) use image buffer */
#define codecFlagUseScreenBuffer (1L<<1)
                                /* (input) use screen buffer */
#define codecFlagUpdatePrevious  (1L<<2)
                                /* (input) update previous buffer */
#define codecFlagNoScreenUpdate  (1L<<3)
                                /* (input) don't update screen */
#define codecFlagWasCompressed   (1L<<4)
                                /*(input) image was compressed */
#define codecFlagDontOffscreen   (1L<<5)
                                /* don't go offscreen */
```

```
#define codecFlagUpdatePreviousComp (1L<<6)
                                /* (input) update previous buffer */
#define codecFlagForceKeyFrame (1L<<7)
                                /* force key frame from image */
#define codecFlagOnlyScreenUpdate
                             (1L<<8)
                                /* (input) only update screen */
#define  codecFlagLiveGrab (1L<<9)
                                /* (input) grab live video */
#define  codecFlagUsedNewImageBuffer
                             (1L<<14)
                                /* (output) new image buffer used */
#define codecFlagUsedImageBuffer
                             (1L<<15)
                             /* (output) decompressor used
                                 offscreen buffer */
```

Constant descriptions

`codecFlagUseImageBuffer`
: Controls whether the decompressor allocates an offscreen buffer for decompression. If your application sets this flag to 1, the decompressor allocates an offscreen buffer the size of the compressed image. If you set this flag to 0, the decompressor does not use an offscreen image buffer. These image buffers are useful when decompressing sequences that were created using temporal compression. For more information about image buffers, see “Using Screen Buffers and Image Buffers” on page 3-34.

`codecFlagUseScreenBuffer`
: Controls whether the decompressor allocates an offscreen destination buffer during decompression. If you set this flag to 1, the decompressor allocates an offscreen buffer the size of the destination screen. If you set this flag to 0, the decompressor does not use an offscreen screen buffer. Using a screen buffer helps to reduce tearing that may result when decompressing directly to the screen. For more information about screen buffers, see “Using Screen Buffers and Image Buffers” on page 3-34.

`codecFlagUpdatePrevious`
: Controls whether the compressor updates the previous image buffer during compression. This flag is only used with sequences that are being temporally compressed. If you set this flag to 1, the compressor copies the current source image into the previous frame buffer at the end of the frame compression.

`codecFlagNoScreenUpdate`
: Controls whether the decompressor updates the screen image. If you set this flag to 1, the decompressor does not write the current frame to the screen, but does write the frame to its offscreen image buffer (if one was allocated). If you set this flag to 0, the decompressor writes the frame to the screen.

`codecFlagWasCompressed`
: Indicates to the compressor that the image to be compressed has been compressed before. This information may be useful to compressors that can compensate for the image degradation that may otherwise result from repeated compression and decompression of the same image. Set this flag to 1 to indicate that the image was previously compressed. Set this flag to 0 if the image was not previously compressed.

`codecFlagDontOffscreen`
: Controls whether the decompressor uses the offscreen buffer during sequence decompression. This flag is only used with sequences that have been temporally compressed. If this flag is set to 1, the decompressor does not use the offscreen buffer during decompression. Instead, the decompressor returns an error. This allows your application to refill the offscreen buffer. If this flag is set to 0, the decompressor uses the offscreen buffer if appropriate.

`codecFlagUpdatePreviousComp`
: Controls whether the compressor updates the previous image buffer with the decompressed image data. This flag is only used with temporal compression and is similar to the `codecFlagUpdatePrevious` flag. As with the `codecFlagUpdatePrevious` flag, if you set this flag to 1, the compressor updates the previous frame buffer at the end of the frame compression. However, this flag causes the Image Compression Manager to update the frame buffer using an image obtained by decompressing the results of the most recent compression operation, rather than the source image.

`codecFlagForceKeyFrame`
: Controls whether the compressor creates a key frame from the current image. This flag is only used with temporal compression. If you set this flag to 1, the compressor makes the current image a key frame. If you set this flag to 0, the compressor decides based on other criteria, such as the key frame rate, whether to create a key frame from the current image.

`codecFlagOnlyScreenUpdate`
: Controls whether the decompressor decompresses the current frame. If you set this flag to 1, the decompressor writes the contents of its offscreen image buffer to the screen, but does decompress the current frame. If you set this flag to 0, the decompressor decompresses the current frame and writes it to the screen. You can set this flag to 1 only if you have allocated an offscreen image buffer for use by the decompressor.

`codecFlagLiveGrab`
: Indicates to the compressor whether the current sequence results from grabbing live video. When working with live video, compressors operate as quickly as possible and disable some additional processing, such as compensation for previously compressed data. Set this flag to 1 when you are compressing from a live video source—the compressor then operates as quickly as it can.

`codecFlagUsedNewImageBuffer`
: Indicates to your application that the decompressor used the offscreen image buffer for the first time when it processed this frame. If this flag is set to 1, the decompressor used the image buffer for this frame and this is the first time the decompressor used the image buffer in this sequence. If this flag is set to 0, the decompressor did not use the image buffer.

`codecFlagUsedImageBuffer`
: Indicates to your application that the decompressor used the offscreen image buffer for this frame. If this flag is set to 1, the decompressor used the image buffer. If this flag is set to 0, the decompressor did not use the image buffer.

Image Compression Manager Functions

The following sections describe the functions that the Image Compression Manager provides to application programs. This section is divided into the following topics:

- "Getting Information About Compressor Components" describes the Image Compression Manager functions that allow applications to gather information about the manager and installed compressor components
- "Getting Information About Compressed Data" describes the functions that allow applications to obtain information about compressed images
- "Working With Images" defines the functions that applications can use to compress and decompress single-frame images that are stored in pixel maps
- "Working With Pictures and PICT Files" describes the functions that applications can use to compress and decompress single-frame images that are stored as pictures or picture files (PICT files)
- "Making Thumbnail Pictures" defines the functions that create and manipulate thumbnail images
- "Working With Sequences" describes the functions that allow applications to compress and decompress sequences of images
- "Changing Sequence-Compression Parameters" discusses the functions that your application can use to manipulate many of the parameters that govern sequence-compression operations
- "Constraining Compressed Data" describes the functions and a data structure that allow your application to communicate information to compressors that can constrain compressed data to a specific data rate

- "Changing Sequence-Decompression Parameters" discusses the functions that your application can use to manipulate many of the parameters that govern sequence-decompression operations
- "Working With the StdPix Function" describes the functions that work with the `StdPix` function to allow your application to have access to compressed image data as it is displayed
- "Aligning Windows" describes the functions that your application can use to position individual windows along optimal alignment grids
- "Working With Graphics Devices and Graphics Worlds" discusses the functions that let you select graphics devices and specify graphics worlds for use in compression and decompression operations

Getting Information About Compressor Components

This section describes the functions that allow your application to gather information about the Image Compression Manager and the installed compressor components.

You can use the `CodecManagerVersion` function to retrieve the version number associated with the Image Compression Manager that is installed on a particular computer.

You can use the `FindCodec`, `GetCodecInfo`, and `GetCodecNameList` functions to locate and retrieve information about the compressor components that are available on a computer.

CodecManagerVersion

Your application can determine the version of the installed Image Compression Manager by calling the `CodecManagerVersion` function.

```
pascal OSErr CodecManagerVersion (long *version);
```

`version` Contains a pointer to a long integer that is to receive the version information. The Image Compression Manager returns its version number into this location. The version number is a long integer value.

DESCRIPTION

The `CodecManagerVersion` function returns the version information as a long integer value.

RESULT CODES

noErr	0	No error

SEE ALSO

"Getting Information About Compressors and Compressed Data," which begins on page 3-24, describes how to use CodecManagerVersion.

GetCodecNameList

The GetCodecNameList function allows your application to retrieve a list of installed compressor components or a list of installed compressor types. This information may be useful when the user selects a compression type for a given image or sequence.

```
pascal OSErr GetCodecNameList (CodecNameSpecListPtr *list,
                               short showAll);
```

list — Contains a pointer to a field that is to receive a pointer to the compressor name list structure. The Image Compression Manager creates the appropriate list and returns a pointer to that list in the field specified by the list parameter. Note that the GetCodecNameList function creates this list in your application's current heap zone.

showAll — Specifies a short integer that controls the contents of the list. Set this parameter to 1 to receive a list of the names of all installed compressor components—the returned list contains one entry for each installed compressor. Set this parameter to 0 to receive a list of the types of installed compressor components—the returned list contains one entry for each installed compressor type. See "The Compressor Name List Structure" on page 3-56 for a complete description of the contents of the returned list.

DESCRIPTION

The Image Compression Manager returns this information in a compressor name list structure, which contains an array of compressor name structures and a field indicating the number of structures in the array.

The CodecType data type defines a field in the compressor name list structure that identifies the compression method employed by a given compressor component. Apple Computer's Developer Technical Support group assigns these values so that they remain unique. These values correspond, in turn, to text strings that can identify the compression method to the user.

```
typedef long CodecType;
/* compressor type descriptor--for example 'jpeg','rle ',
    'rpza' */
```

Currently, six CodecType values are provided by Apple. You should use the GetCodecNameList function to retrieve these names, so that your application can take advantage of new compressor types that may be added in the future. For each CodecType value in the following list, the corresponding compression method is also identified by its text string name. For more information about each of these compression techniques, see the section "About Image Compression," which begins on page 3-8.

Table 3-3 Compressor type descriptors

Compressor type	Compressor name
'rpza'	video compressor
'jpeg'	photo compressor
'rle '	animation compressor
'raw '	raw compressor
'smc '	graphics compressor
'cdvc'	compact video compressor

SPECIAL CONSIDERATIONS

Note that the Image Compression Manager returns the list in your application's current heap zone. Use the DisposeCodecNameList function, described in the next section, to release this memory when your program is finished with the list.

RESULT CODES

noErr	0	No error
paramErr	–50	Invalid parameter specified

Component Manager errors
Memory Manager errors

DisposeCodecNameList

The DisposeCodecNameList function allows your application to dispose of the compressor name list structure you obtained by calling the GetCodecNameList function.

```
pascal OSErr DisposeCodecNameList (CodecNameSpecListPtr list);
```

`list` Points to the compressor name list to be disposed of. You obtain the compressor list by calling the `GetCodecNameList` function, which is described in the previous section.

RESULT CODES

`noErr`	0	No error

Memory Manager errors

GetCodecInfo

The `GetCodecInfo` function returns information about a single compressor component.

```
pascal OSErr GetCodecInfo (CodecInfo *info, CodecType cType,
                           CodecComponent codec);
```

`info` Contains a pointer to a compressor information structure. The `GetCodecInfo` function returns the detailed information about the appropriate compressor component into this structure.

`cType` Specifies a compressor type. You must set this parameter to a valid compressor type (see Table 3-3 on page 3-64 for a list of the available compressor types).

`codec` Specifies a compressor identifier. Set this parameter to the component identifier of the specific compressor for the request. The component identifier is available in the compressor name list structure returned by the `GetCodecNameList` function (described on page 3-63).

If you want information about any compressor of the type specified by the `cType` parameter, set `codec` to 0. The Image Compression Manager then returns information about the first compressor it finds of the type you have specified.

DESCRIPTION

Your application may retrieve information about a specific compressor or about a compressor of a specific type. If you request information about a type of compressor, the Image Compression Manager returns information about the first compressor it finds of that type. The Image Compression Manager returns the detailed compressor information in a compressor information structure (see “The Compressor Information Structure,” which begins on page 3-52, for details).

RESULT CODES

noErr	0	No error
noCodecErr	–8961	The Image Compression Manager could not find the specified compressor

Component Manager errors
Memory Manager errors

FindCodec

The FindCodec function allows you to determine which of the installed compressors or decompressors has been chosen to field requests made using one of the special compressor identifiers.

Some Image Compression Manager functions allow you to specify a particular compressor component. For example, you may use the codec parameter to the CompressSequenceBegin function (described on page 3-106) to specify a particular compressor to do the compression.

You identify the compressor to the Image Compression Manager by specifying the compressor's component identifier (see the description of the GetCodecNameList function on page 3-63 for information on retrieving these identifiers).

The Image Compression Manager also supports several special identifiers that allow you to exert some control over the component for a given action without having to know its identifier.

```
pascal OSErr FindCodec (CodecType cType, CodecComponent specCodec,
                        CompressorComponent *compressor,
                        DecompressorComponent *decompressor);
```

cType Specifies a compressor type. You must set this parameter to a valid compressor type (see Table 3-3 on page 3-64 for a list of the available compressor types).

specCodec Contains a special identifier value. You must set this parameter to one of the following special identifier values:

anyCodec Choose the first compressor or decompressor of the specified type

bestSpeedCodec Choose the fastest compressor or decompressor of the specified type

bestFidelityCodec Choose the most accurate compressor or decompressor of the specified type

bestCompressionCodec Choose the compressor that produces the smallest resulting data

`compressor`

Contains a pointer to a field to receive the identifier for the compressor component. The Image Compression Manager returns the identifier of the compressor that meets the special characteristics you specify in the `specCodec` parameter. Note that this identifier may differ from the value of the field referred to by the `decompressor` field. The Image Compression Manager sets this field to 0 if it cannot find a suitable compressor component. Set this parameter to `nil` if you do not want this information.

`decompressor`

Contains a pointer to a field to receive the identifier for the decompressor component. The Image Compression Manager returns the identifier of the decompressor that meets the special characteristics you specify in the `specCodec` parameter. Note that this identifier may differ from the value of the field referred to by the `compressor` field. The Image Compression Manager sets this field to 0 if it cannot find a suitable decompressor component. Set this parameter to `nil` if you do not want this information.

DESCRIPTION

You can use the `FindCodec` function to obtain the identifier of the component that is being used to field requests made with one of the special compressor identifiers.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memFullErr`	–108	Not enough memory available
`noCodecErr`	–8961	The Image Compression Manager could not find the specified compressor

Getting Information About Compressed Data

This section describes the functions that enable your application to collect information about compressed images and images that are about to be compressed. Your application may use some of these functions in preparation for compressing or decompressing an image or sequence.

You can use the `GetCompressionTime` function to determine how long it will take for a compressor to compress a specified image. Similarly, you can use the `GetMaxCompressionSize` function to find out how large the compressed image may be after the compression operation.

You can use the `GetCompressedImageSize` to determine the size of a compressed image that does not have a complete image description.

The `GetSimilarity` function allows you to determine how similar two images are. This information is useful when you are performing temporal compression on an image sequence.

GetMaxCompressionSize

The `GetMaxCompressionSize` function allows your application to determine the maximum size an image will be after compression. You specify the compression characteristics, including compression type and quality, along with the image.

```
pascal OSErr GetMaxCompressionSize (PixMapHandle src,
                                    const Rect *srcRect,
                                    short colorDepth,
                                    CodecQ quality,
                                  CodecType cType,
                                  CompressorComponent codec,
                                  long *size);
```

`src` Contains a handle to the source image. The source image must be stored in a pixel map structure. The compressor uses only the image's size and pixel depth to determine the maximum size of the compressed image.

`srcRect` Contains a pointer to a rectangle defining the portion of the source image that is to be compressed. You may set this parameter to `nil` if you are interested only in information about quality settings. `GetCompressionTime` then uses the bounds of the source pixel map.

`colorDepth`

Specifies the depth at which the image is to be compressed. If you set this parameter to 0, the Image Compression Manager determines the appropriate value for the source image. Values of 1, 2, 4, 8, 16, 24, and 32 indicate the number of bits per pixel for color images. Values of 34, 36, and 40 indicate 2-bit, 4-bit, and 8-bit grayscale, respectively, for grayscale images. Your program can determine which depths are supported by a given compressor by examining the compressor information structure returned by the `GetCodecInfo` function (see "Getting Information About Compressor Components" on page 3-62 for more information on the `GetCodecInfo` function).

`quality` Specifies the desired compressed image quality. See "Compression Quality Constants" beginning on page 3-57 for valid values.

`cType` Specifies a compressor type. You must set this parameter to a valid compressor type (see Table 3-3 on page 3-64 for a list of the available compressor types).

`codec` Contains a compressor identifier. Specify a particular compressor by setting this parameter to its compressor identifier. Alternatively, you may use one of the special identifiers:

`anyCodec` Choose the first compressor of the specified type

`bestSpeedCodec`
Choose the fastest compressor of the specified type

`bestFidelityCodec`
Choose the most accurate compressor of the specified type

`bestCompressionCodec`
Choose the compressor that produces the smallest resulting data

You can also specify a component instance. This may be useful if you have previously set some parameter on a specific instance of a `codec` field and want to make sure that the specified instance is used for that operation.

`size` Contains a pointer to a field to receive the size, in bytes, of the compressed image.

DESCRIPTION

The Image Compression Manager returns the maximum resulting size for the specified image and parameters. Your application may then use this information to allocate memory for the compression operation.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`noCodecErr`	–8961	The Image Compression Manager could not find the specified compressor

GetCompressionTime

The `GetCompressionTime` function allows your application to determine the estimated amount of time required to compress a given image. This function also allows you to verify that the quality settings you desire are supported by a given compressor component.

You specify the compression characteristics, including compression type and quality, along with the image.

```
pascal OSErr GetCompressionTime (PixMapHandle src,
                                 const Rect *srcRect,
                                 short colorDepth,
                                 CodecType cType,
                                 CompressorComponent codec,
                                 CodecQ *spatialQuality,
                                 CodecQ *temporalQuality,
                                 unsigned long *compressTime);
```

`src` Contains a handle to the source image. The source image must be stored in a pixel map structure. The compressor uses only the bit depth of this image to determine the compression time. You may set this parameter to `nil` if you are interested only in information about quality settings.

`srcRect` Contains a pointer to a rectangle defining the portion of the source image to compress. You may set this parameter to `nil` if you are interested only in information about quality settings. `GetCompressionTime` then uses the bounds of the source pixel map.

`colorDepth`
Specifies the depth at which the image is to be compressed. If you set this parameter to 0, the Image Compression Manager determines the appropriate value for the source image. Values of 1, 2, 4, 8, 16, 24, and 32 indicate the number of bits per pixel for color images. Values of 34, 36, and 40 indicate 2-bit, 4-bit, and 8-bit grayscale, respectively, for grayscale images. Your program can determine which depths are supported by a given compressor by examining the compressor information structure returned by the `GetCodecInfo` function (see "Getting Information About Compressor Components" on page 3-62 for more information on the `GetCodecInfo` function).

`cType` Specifies a compressor type. You must set this parameter to a valid compressor type (see Table 3-3 on page 3-64 for a list of the available compressor types).

`codec` Contains a compressor identifier. Specify a particular compressor by setting this parameter to its compressor identifier. Alternatively, you may use one of the special identifiers:

`anyCodec` Choose the first compressor of the specified type

`bestSpeedCodec`
Choose the fastest compressor of the specified type

`bestFidelityCodec`
Choose the most accurate compressor of the specified type

`bestCompressionCodec`
Choose the compressor that produces the smallest resulting data

You can also specify a component instance. This may be useful if you have previously set some parameter on a specific instance of a `codec` field and want to make sure that the specified instance is used for that operation.

`spatialQuality`
Contains a pointer to a field containing the desired compressed image quality. The Image Compression Manager sets this field to the closest actual quality that the compressor can achieve. See "Compression Quality Constants" beginning on page 3-57 for valid values. If you are not interested in this information, pass `nil` in this parameter.

`temporalQuality`
Contains a pointer to a field containing the desired temporal quality. Use this value only with images that are part of image sequences. The Image Compression Manager sets this field to the closest actual quality that the compressor can achieve. See "Compression Quality Constants" beginning on page 3-57 for valid values. If you are not interested in this information, pass `nil` in this parameter.

`compressTime`
Contains a pointer to a field to receive the compression time, in milliseconds. If the compressor cannot determine the amount of time required to compress the image or if the compressor does not support this function, this field is set to 0. If you are not interested in this information, pass `nil` in this parameter.

DESCRIPTION

The Image Compression Manager returns the maximum compression time for the specified image and parameters. Note that some compressors may not support this function. If the component you specify does not support this function, the Image Compression Manager returns a time value of 0.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`noCodecErr`	–8961	The Image Compression Manager could not find the specified compressor

GetSimilarity

The `GetSimilarity` function compares a compressed image to a picture stored in a pixel map and returns a value indicating the relative similarity of the two images.

```
pascal OSErr GetSimilarity (PixMapHandle src, const Rect *srcRect,
                            ImageDescriptionHandle desc, Ptr data,
                            Fixed *similarity);
```

`src` Contains a handle to the noncompressed image. The image must be stored in a pixel map structure.

`srcRect` Contains a pointer to a rectangle defining the portion of the image to compare to the compressed image. This rectangle should be the same size as the image described by the image description structure specified by the `desc` parameter.

`desc` Specifies a handle to the image description structure that defines the compressed image for the operation.

`data` Points to the compressed image data. This pointer must contain a 32-bit clean address.

`similarity`
Contains a pointer to a field that is to receive the similarity value. The compressor sets this field to reflect the relative similarity of the two images. Valid values range from 0 (completely different) to 1.0 (identical).

RESULT CODES

noErr	0	No error
paramErr	–50	Invalid parameter specified
memFullErr	–108	Not enough memory available
noCodecErr	–8961	The Image Compression Manager could not find the specified compressor

GetCompressedImageSize

The `GetCompressedImageSize` function determines the size, in bytes, of a compressed image.

Most applications do not need to use this function because compressed images have a corresponding image description structure with a size field. You only need to use this function if you do not have an image description structure associated with your data—for example, when you are taking a compressed image out of a movie one frame at a time.

```
pascal OSErr GetCompressedImageSize (ImageDescriptionHandle desc,
                                     Ptr data, long bufferSize,
                                     DataProcRecordPtr dataProc,
                                     long *dataSize);
```

`desc` Specifies a handle to the image description structure that defines the compressed image for the operation.

`data` Points to the compressed image data. This pointer must contain a 32-bit clean address.

`bufferSize` Specifies the size of the buffer to be used by the data-loading function specified by the `dataProc` parameter. If you have not specified a data-loading function, set this parameter to 0.

`dataProc` Points to a data-loading function structure. If the data stream is not all in memory when your program calls `GetCompressedImageSize`, the compressor calls a function you provide that loads more compressed data (see "Data-Loading Functions" beginning on page 3-149 for more information about data-loading functions). If you have not provided a data-loading function, set this parameter to `nil`. In this case, the entire image must be in memory at the location specified by the `data` parameter.

`dataSize` Contains a pointer to a field that is to receive the size, in bytes, of the compressed image.

DESCRIPTION

Your application may use the `GetCompressedImageSize` function when parsing a data stream that does not contain an image description structure for each frame in the sequence.

RESULT CODES

noErr	0	No error
paramErr	–50	Invalid parameter specified
noCodecErr	–8961	Image Compression Manager could not find the specified compressor
codecSpoolErr	–8966	Error loading or unloading data

Component Manager errors

Working With Images

This section discusses the functions that allow your application to compress and decompress single-frame images stored as pixel maps (of data type `PixMap`). See "Working With Sequences," which begins on page 3-106, for information on compressing and decompressing sequences of images. See "Working With Pictures and PICT Files," which begins on page 3-88, for information on compressing and manipulating single-frame images stored as pictures or picture files (in PICT format).

The Image Compression Manager provides two sets of functions for compressing and decompressing images. If you do not need to assert a lot of control over the compression operation, you can use the `CompressImage` and `DecompressImage` functions to work with compressed images. If you need more control over the compression parameters, you can use the `FCompressImage` and `FDecompressImage` functions.

You can convert a compressed image from one compression format to another by calling the `ConvertImage` function.

You can alter the spatial characteristics of a compressed image by calling the `TrimImage` function.

You can work with an image's color table with the `SetImageDescriptionCTable` and `GetImageDescriptionCTable` functions.

CompressImage

The `CompressImage` function allows your application to compress a single-frame image that is currently stored as a pixel map structure.

```
pascal OSErr CompressImage (PixMapHandle src,
                            const  Rect *srcRect,
                            CodecQ quality, CodecType cType,
                            ImageDescriptionHandle desc,
                            Ptr data);
```

`src` Contains a handle to the image to be compressed. The image must be stored in a pixel map structure.

`srcRect` Contains a pointer to a rectangle defining the portion of the image to compress.

`quality` Specifies the desired compressed image quality. See "Compression Quality Constants" on page 3-57 for valid values.

`cType` Specifies a compressor type. You must set this parameter to a valid compressor type (see Table 3-3 on page 3-64 for a list of the available compressor types).

`desc` Contains a handle that is to receive a formatted image description structure. The Image Compression Manager resizes this handle for the returned image description structure. Your application should store this image description with the compressed image data.

`data` Points to a location to receive the compressed image data. It is your program's responsibility to make sure that this location can receive at least as much data as indicated by the `GetMaxCompressionSize` function (described on page 3-68). The Image Compression Manager places the actual size of the compressed image into the `dataSize` field of the image description structure referred to by the `desc` parameter. This pointer must contain a 32-bit clean address. If you use a dereferenced, locked handle, you must call the Memory Manager's `StripAddress` routine before you use that handle with this parameter (for details on `StripAddress`, see *Inside Macintosh: Memory*).

DESCRIPTION

The `CompressImage` function presents a simplified interface to your application, eliminating some parameters for the sake of convenience.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memFullErr`	–108	Not enough memory available
`noCodecErr`	–8961	The Image Compression Manager could not find the specified compressor

SEE ALSO

If you need to exert greater control over the compression operation, use the `FCompressImage` function, described in the next section.

FCompressImage

Like the `CompressImage` function, the `FCompressImage` function allows your application to compress a single-frame image that is currently stored as a pixel map structure (`PixMap`).

```
pascal OSErr FCompressImage (PixMapHandle src,
                             const  Rect *srcRect,
                             short colorDepth, CodecQ quality,
                             CodecType cType,
                             CompressorComponent codec,
                             CTabHandle clut, CodecFlags flags,
                             long bufferSize,
                             FlushProcRecordPtr flushProc,
                             ProgressProcRecordPtr progressProc,
                             ImageDescriptionHandle desc,
                             Ptr data);
```

`src` Contains a handle to the image to be compressed. The image must be stored in a pixel map structure.

`srcRect` Contains a pointer to a rectangle defining the portion of the image to compress.

`colorDepth`

Specifies the depth at which the image is likely to be viewed. Compressors may use this as an indication of the color or grayscale resolution of the compressed image. If you set this parameter to 0, the Image Compression Manager determines the appropriate value for the source image. Values of 1, 2, 4, 8, 16, 24, and 32 indicate the number of bits per pixel for color images. Values of 34, 36, and 40 indicate 2-bit, 4-bit, and 8-bit grayscale, respectively, for grayscale images. Your program can determine which depths are supported by a given compressor by examining the compressor information structure returned by the `GetCodecInfo` function (see “Getting Information About Compressor Components” on page 3-62 for more information on the `GetCodecInfo` function).

`quality` Specifies the desired compressed image quality. See “Compression Quality Constants” on page 3-57, for valid values.

`cType` Specifies a compressor type. You must set this parameter to a valid compressor type (see Table 3-3 on page 3-64 for a list of the available compressor types).

`codec` Specifies a compressor identifier. Specify a particular compressor by setting this parameter to its compressor identifier. Alternatively, you may use one of the special identifiers:

`anyCodec` Choose the first compressor of the specified type

`bestSpeedCodec`
Choose the fastest compressor of the specified type

`bestFidelityCode`
Choose the most accurate compressor of the specified type

`bestCompressionCodec`
Choose the compressor that produces the smallest resulting data

You can also specify a component instance. This may be useful if you have previously set some parameter on a specific instance of a `codec` field and want to make sure that the specified instance is used for that operation.

If you set the `codec` parameter to `anyCodec`, the Image Compression Manager chooses the first compressor it finds of the specified type.

`clut` Contains a handle to a custom color lookup table. Your program may use this parameter to indicate a custom color lookup table to be used with this image. If the value of the `colorDepth` parameter is less than or equal to 8 and the custom color lookup table is different from that of the source pixel map (that is, the `ctSeed` field values differ in the two pixel maps), the compressor remaps the colors of the image to the custom colors. If you set the `colorDepth` parameter to 16, 24, or 32, the compressor stores the custom color table with the compressed image. The compressor may use the table to specify the best colors to use when displaying the image at lower bit depths. The compressor ignores the `clut` parameter when `colorDepth` is set to 33, 34, 36, or 40. If you set this parameter to `nil`, the compressor uses the color lookup table from the source pixel map.

`flags` Specifies flags providing further control information. See “Image Compression Manager Function Control Flags,” which begins on page 3-58, for information about `CodecFlags` fields. The following flag is available for this function:

`codecFlagWasCompressed`
Indicates to the compressor that the image to be compressed has been compressed before. This information may be useful to compressors that can compensate for the image degradation that may otherwise result from repeated compression and decompression of the same image. Set this flag to 1 to indicate that the image was previously compressed. Set this flag to 0 if the image was not previously compressed.

`bufferSize`
: Specifies the size of the buffer to be used by the data-unloading function specified by the `flushProc` parameter. If you have not specified a data-unloading function, set this parameter to 0.

`flushProc`
: Points to a data-unloading function structure. If there is not enough memory to store the compressed image, the compressor calls a function you provide that unloads some of the compressed data (see "Data-Unloading Functions" beginning on page 3-150 for more information on the data-unloading structure). If you have not provided a data-unloading function, set this parameter to `nil`. In this case, the compressor writes the entire compressed image into the memory location specified by the `data` parameter.

`progressProc`
: Points to a progress function structure. During the compression operation, the compressor may occasionally call a function you provide in order to report its progress (see "Progress Functions" on page 3-152 for more information about progress functions). If you have not provided a progress function, set this parameter to `nil`. If you pass a value of –1, you obtain a standard progress function.

`desc`
: Contains a handle that is to receive a formatted image description structure. The Image Compression Manager resizes this handle for the returned image description structure. Your application should store this image description with the compressed image data.

`data`
: Points to a location to receive the compressed image data. It is your program's responsibility to make sure that this location can receive at least as much data as indicated by the `GetMaxCompressionSize` function (described on page 3-68). If there is not sufficient memory to store the compressed image, you may choose to write the compressed data to mass storage during the compression operation. Use the `flushProc` parameter to identify your data-unloading function to the compressor. This pointer must contain a 32-bit clean address. If you use a dereferenced, locked handle, you must call the Memory Manager's `StripAddress` routine before you use that handle with this parameter. (See *Inside Macintosh: Memory* for details on `StripAddress`.)

  The Image Compression Manager places the actual size of the compressed image into the `dataSize` field of the image description structure referred to by the `desc` parameter.

DESCRIPTION

The `FCompressImage` function gives your application additional control over the parameters that guide the compression operation.

RESULT CODES

noErr	0	No error
paramErr	–50	Invalid parameter specified
memFullErr	–108	Not enough memory available
noCodecErr	–8961	The Image Compression Manager could not find the specified compressor
codecSpoolErr	–8966	Error loading or unloading data
codecAbortErr	–8967	Operation aborted by the progress function

SEE ALSO

If you find that you do not need this level of compression parameter control, use the `CompressImage` function, described in the previous section.

DecompressImage

The `DecompressImage` function allows your application to decompress a single-frame image into a pixel map structure. If you call this function when you have a picture open, the Image Compression Manager inserts the compressed image data into the picture.

```
pascal OSErr DecompressImage (Ptr data,
                        ImageDescriptionHandle desc,
                        PixMapHandle dst, const Rect *srcRect,
                        const  Rect *dstRect, short mode,
                        RgnHandle mask);
```

`data` Points to the compressed image data. This pointer must contain a 32-bit clean address. If you use a dereferenced, locked handle, you must call the Memory Manager's `StripAddress` routine before you use that handle with this parameter.

`desc` Contains a handle to the image description structure that describes the compressed image.

`dst` Contains a handle to the pixel map where the decompressed image is to be displayed. Set the current graphics port to the port that contains this pixel map.

`srcRect` Contains a pointer to a rectangle defining the portion of the image to decompress. This rectangle must lie within the boundary rectangle of the compressed image, which is defined by (0,0) and `((**desc).width,(**desc).height)`. If you want to decompress the entire source image, set this parameter to `nil`. If the parameter is `nil`, the rectangle is set to the rectangle structure of the image description structure.

`dstRect`	Contains a pointer to the rectangle into which the decompressed image is to be loaded. The compressor scales the source image to fit into this destination rectangle.
`mode`	Specifies the transfer mode for the operation. The Image Compression Manager supports the same transfer modes supported by QuickDraw's `CopyBits` routine.
`mask`	Contains a handle to a clipping region in the destination coordinate system. If specified, the compressor applies this mask to the destination image. If you do not want to mask bits in the destination, set this parameter to `nil`.

DESCRIPTION

The `DecompressImage` function presents a simplified interface to your application, eliminating some parameters for the sake of convenience.

Note that the `DecompressImage` function is invoked through the `StdPix` function (see "Working With the StdPix Function," which begins on page 3-137, for more information).

SPECIAL CONSIDERATIONS

The graphics port and the graphics device should be set to the destination before you call the `DecompressImage` function.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memFullErr`	–108	Not enough memory available
`noCodecErr`	–8961	The Image Compression Manager could not find the specified compressor

SEE ALSO

If you need to exert greater control over the decompression operation, use the `FDecompressImage` function, described in the next section.

FDecompressImage

Like the `DecompressImage` function, the `FDecompressImage` function allows your application to decompress a single-frame image into a pixel map (`PixMap`). This function gives your application greater control over the parameters that guide the

decompression operation. If you find that you do not need this level of control, use the `DecompressImage` function, described in the previous section.

```
pascal OSErr FDecompressImage (Ptr data,
                               ImageDescriptionHandle desc,
                               PixMapHandle dst,
                               const  Rect *srcRect,
                               MatrixRecordPtr matrix,
                               short mode, RgnHandle mask,
                               PixMapHandle matte,
                               const  Rect *matteRect,
                               CodecQ accuracy,
                               DecompressorComponent codec,
                               long bufferSize,
                               DataProcRecordPtr dataProc,
                               ProgressProcRecordPtr progressProc);
```

`data` Points to the compressed image data. If the entire compressed image cannot be stored at this location, your application may provide a data-loading function (see the discussion of the `dataProc` parameter to this function). This pointer must contain a 32-bit clean address. If you use a dereferenced, locked handle, you must call the Memory Manager's `StripAddress` routine before you use that handle with this parameter. (See *Inside Macintosh: Memory* for details on `StripAddress`.)

`desc` Contains a handle to the image description structure that describes the compressed image.

`dst` Contains a handle to the pixel map where the decompressed image is to be displayed. Set the current graphics port to the port that contains this pixel map.

`srcRect` Contains a pointer to a rectangle defining the portion of the image to decompress. This rectangle must lie within the boundary rectangle of the compressed image, which is defined by (0,0) and `((**desc).width,(**desc).height)`. If you want to decompress the entire source image, set this parameter to `nil`. If the parameter is `nil`, the rectangle is set to the rectangle structure of the image description structure.

`matrix` Points to a matrix structure that specifies how to transform the image during decompression. You can use the matrix structure to translate or scale the image during decompression. If you do not want to apply such effects, set the `matrix` parameter to `nil`. See the chapter "Movie Toolbox" in this book for more information about matrix operations.

`mode` Specifies the transfer mode for the operation. The Image Compression Manager supports the same transfer modes supported by QuickDraw's `CopyBits` routine (described in *Inside Macintosh: Imaging*).

`mask` Contains a handle to a clipping region in the destination coordinate system. If specified, the decompressor applies this mask to the destination image. If you do not want to mask bits in the destination, set this parameter to `nil`.

`matte` Contains a handle to a pixel map that contains a blend matte. You can use the blend matte to cause the decompressed image to be blended into the destination pixel map. The matte can be defined at any supported pixel depth—the matte depth need not correspond to the source or destination depths. However, the matte must be in the coordinate system of the source image. If you do not want to apply a blend matte, set this parameter to `nil`.

`matteRect` Contains a pointer to a rectangle defining a portion of the blend matte to apply. If you do not want to use the entire matte referred to by the `matte` parameter, use this parameter to specify a rectangle within that matte. If specified, this rectangle must be the same size as the rectangle specified by the `srcRect` parameter. If you want to use the entire matte, or if you are not providing a blend matte, set this parameter to `nil`.

`accuracy` Specifies the accuracy desired in the decompressed image. Values for this parameter are on the same scale as compression quality. See "Compression Quality Constants" beginning on page 3-57 for valid values. (For a good display of still images, you should specify at least the `codecHighQuality` constant.)

`codec` Contains a compressor identifier. Specify a particular decompressor by setting this parameter to its compressor identifier. Alternatively, you may use one of the special identifiers:

`anyCodec` Choose the first decompressor of the specified type

`bestSpeedCodec` Choose the fastest decompressor of the specified type

`bestFidelityCodec` Choose the most accurate decompressor of the specified type

`bestCompressionCodec` Choose the decompressor that produces the smallest resulting data

You can also specify a component instance. This may be useful if you have previously set some parameter on a specific instance of a `codec` field and want to make sure that the specified instance is used for that operation.

If you set the `codec` parameter to `anyCodec`, the Image Compression Manager chooses the first decompressor it finds of the specified type.

`bufferSize` Specifies the size of the buffer to be used by the data-loading function specified by the `dataProc` parameter. If you have not specified a data-loading function, set this parameter to 0.

`dataProc` Points to a data-loading function structure. If there is not enough memory to store the compressed image, the compressor calls a function you provide that loads more compressed data (see "Data-Loading Functions" on page 3-149 for more information about data-loading functions). If you have not provided a data-loading function, set this parameter to `nil`. In this case, the compressor expects that the entire compressed image is in the memory location specified by the `data` parameter.

`progressProc`
Points to a progress function structure. During the decompression operation, the compressor may occasionally call a function you provide in order to report its progress (see "Progress Functions" on page 3-152 for more information about progress functions). If you have not provided a progress function, set this parameter to `nil`. If you pass a value of –1, you obtain a standard progress function.

DESCRIPTION

Note that this function is invoked through the `StdPix` function (see "Working With the StdPix Function," which begins on page 3-137, for more information).

SPECIAL CONSIDERATIONS

The graphics port and the graphics device should be set to the destination before you call the `FDecompressImage` function.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memFullErr`	–108	Not enough memory available
`noCodecErr`	–8961	The Image Compression Manager could not find the specified compressor
`codecSpoolErr`	–8966	Error loading or unloading data
`codecAbortErr`	–8967	Operation aborted by the progress function

ConvertImage

The `ConvertImage` function allows your application to convert the format of a compressed image. This function is essentially equivalent to decompressing and recompressing the image.

```
pascal OSErr ConvertImage (ImageDescriptionHandle srcDD,
                           Ptr srcData, short colorDepth,
                           CTabHandle clut, CodecQ accuracy,
                           CodecQ quality, CodecType cType,
                           CodecComponent codec,
                           ImageDescriptionHandle dstDD,
                           Ptr dstData);
```

`srcDD` Contains a handle to the image description structure that describes the compressed image.

`srcData` Points to the compressed image data. This pointer must contain a 32-bit clean address.

`colorDepth`

Specifies the depth at which the recompressed image is likely to be viewed. Decompressors may use this as an indication of the color or grayscale resolution of the compressed image. If you set this parameter to 0, the Image Compression Manager determines the appropriate value for the source image. Values of 1, 2, 4, 8, 16, 24, and 32 indicate the number of bits per pixel for color images. Values of 34, 36, and 40 indicate 2-bit, 4-bit, and 8-bit grayscale, respectively, for grayscale images. Your program can determine which depths are supported by a given compressor by examining the compressor information structure returned by the `GetCodecInfo` function (see “Getting Information About Compressor Components” on page 3-62 for more information on the `GetCodecInfo` function).

`clut` Contains a handle to a custom color lookup table. Your program may use this parameter to indicate a custom color lookup table to be used with this image. If the value of the `colorDepth` parameter is less than or equal to 8 and the custom color lookup table is different from that of the source pixel map (that is, the `ctSeed` field values differ in the two pixel maps), the compressor remaps the colors of the image to the custom colors. If you set the `colorDepth` parameter to 16, 24, or 32, the compressor stores the custom color table with the compressed image. The compressor may use the table to specify the best colors to use when displaying the image at lower bit depths. The compressor ignores the `clut` parameter when `colorDepth` is set to 33, 34, 36, or 40. If you set this parameter to `nil`, the compressor uses the color lookup table from the source image description structure.

`accuracy` Specifies the accuracy desired in the decompressed image. Values for this parameter are on the same scale as compression quality. See “Compression Quality Constants” on page 3-57 for valid values. (For a good display of still images, you should specify at least the `codecHighQuality` constant.)

`quality` Specifies the desired compressed image quality. See "Compression Quality Constants" on page 3-57 for valid values. Use the following value:

`codecHighQuality`
Specifies high-quality image reproduction. This value should correspond to the highest image quality that can be achieved with reasonable performance.

`cType` Specifies a compressor type. You must set this parameter to a valid compressor type. See Table 3-3 on page 3-64 for a list of the available compressor types.

`codec` Contains a compressor identifier. Specify a particular compressor by setting this parameter to its compressor identifier. Alternatively, you may use one of the special identifiers:

`anyCodec` Choose the first compressor of the specified type

`bestSpeedCodec`
Choose the fastest compressor of the specified type

`bestFidelityCodec`
Choose the most accurate compressor of the specified type

You can also specify a component instance. This may be useful if you have previously set some parameter on a specific instance of a `codec` field and want to make sure that the specified instance is used for that operation.

If you set the `codec` parameter to `anyCodec`, the Image Compression Manager chooses the first compressor it finds of the specified type.

`dstDD` Contains a handle that is to receive a formatted image description structure. The Image Compression Manager resizes this handle for the returned image description structure. Your application should store this image description with the compressed image data.

`dstData` Points to a location to receive the compressed image data. It is your program's responsibility to make sure that this location can receive at least as much data as indicated by the `GetMaxCompressionSize` function (described on page 3-68). The Image Compression Manager places the actual size of the compressed image into the `dataSize` field of the image description referred to by the `dstDD` parameter. This pointer must contain a 32-bit clean address.

DESCRIPTION

During the decompression operation, the decompressor uses the `srcDD`, `srcData`, and `accuracy` parameters. During the subsequent compression operation, the compressor uses the `colorDepth`, `clut`, `cType`, `codec`, `quality`, `dstDD`, and `dstData` parameters.

RESULT CODES

noErr	0	No error
paramErr	–50	Invalid parameter specified
memFullErr	–108	Not enough memory available
noCodecErr	–8961	The Image Compression Manager could not find the specified compressor

TrimImage

The TrimImage function adjusts a compressed image to the boundaries defined by a rectangle specified by your application.

```
pascal OSErr TrimImage (ImageDescriptionHandle desc, Ptr inData,
                        long inBufferSize,
                        DataProcRecordPtr dataProc,
                        Ptr outData, long outBufferSize,
                        FlushProcRecordPtr flushProc,
                        Rect *trimRect,
                        ProgressProcRecordPtr progressProc);
```

desc Contains a handle to the image description structure that describes the compressed image. On return from TrimImage, the compressor updates this image description to refer to the resized image.

inData Points to the compressed image data. If the entire compressed image cannot be stored at this location, your application may provide a data-loading function (see the discussion of the dataProc parameter to this function). This pointer must contain a 32-bit clean address.

inBufferSize
Specifies the size of the buffer to be used by the data-loading function specified by the dataProc parameter. If you have not specified a data-loading function, this parameter is ignored.

dataProc Points to a data-loading function structure. If there is not enough memory to store the compressed image, the compressor calls a function you provide that loads more compressed data (see “Data-Loading Functions” beginning on page 3-149 for more information about data-loading function structures). If you have not provided a data-loading function, set this parameter to nil. In this case, the compressor expects that the entire compressed image is in the memory location specified by the inData parameter.

outData Points to a buffer to receive the trimmed image. Your application should create this destination buffer at least as large as the source image. If there is not sufficient memory to store the compressed image, you may choose to write the compressed data to mass storage during the compression operation. Use the flushProc parameter to identify your data-unloading function to the compressor. This pointer must contain a 32-bit clean address.

The Image Compression Manager places the actual size of the resulting image into the `dataSize` field of the image description structure referred to by the `desc` parameter.

`outBufferSize`
Specifies the size of the buffer to be used by the data-unloading function specified by the `flushProc` parameter. If you have not specified a data-unloading function, this parameter is ignored.

`flushProc` Points to a data-unloading function structure. If there is not enough memory to store the compressed image, the compressor calls a function you provide that unloads some of the compressed data (see "Data-Unloading Functions" beginning on page 3-150 for more information on the data-unloading structure). If you have not provided a data-unloading function, set this parameter to `nil`. In this case, the compressor writes the entire compressed image into the memory location specified by the `data` parameter.

`trimRect` Contains a pointer to a rectangle that defines the desired image dimensions. Upon return to your application, the compressor adjusts the rectangle values so that they refer to the same rectangle in the result image (this is necessary whenever data is removed from the beginning or from the left side of the image).

`progressProc`
Points to a progress function structure. During the operation, the compressor may occasionally call a function you provide in order to report its progress (see "Progress Functions" on page 3-152 for more information about progress functions). If you have not provided a progress function, set this parameter to `nil`. If you pass a value of –1, you obtain a standard progress function.

DESCRIPTION

The resulting image data is still compressed and is in the same compression format as the source image.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memFullErr`	–108	Not enough memory available
`noCodecErr`	–8961	The Image Compression Manager could not find the specified compressor
`codecUnimpErr`	–8962	Feature not implemented by this compressor
`codecSpoolErr`	–8966	Error loading or unloading data
`codecAbortErr`	–8967	Operation aborted by the progress function

SetImageDescriptionCTable

Your application may use the `SetImageDescriptionCTable` function to update the custom color table for an image. The Image Compression Manager copies the custom color table for an image into the appropriate image description structure. This function does not change the image data, just the color table.

```
pascal OSErr SetImageDescriptionCTable
                            (ImageDescriptionHandle desc,
                               CTabHandle ctable);
```

`desc` Contains a handle to the appropriate image description structure. The `SetImageDescriptionCTable` function updates the size of the image description to accommodate the new color table and removes the old color table, if one is present.

`ctable` Contains a handle to the new color table. The `SetImageDescriptionCTable` function loads this color table into the image description referred to by the `desc` parameter. Set this parameter to `nil` to remove a color table.

DESCRIPTION

The `SetImageDescriptionCTable` function is rarely used. Typically, you supply the color table when your application compresses an image. The Image Compression Manager stores the color table with the image.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memFullErr`	–108	Not enough memory available
`noCodecErr`	–8961	The Image Compression Manager could not find the specified compressor

GetImageDescriptionCTable

Your application may use the `GetImageDescriptionCTable` function to set the custom color table for an image.

```
pascal OSErr GetImageDescriptionCTable
                                (ImageDescriptionHandle desc,
                                 CTabHandle *ctable);
```

`desc` Contains a handle to the appropriate image description structure.

`ctable` Contains a pointer to a field that is to receive a color table handle. The `GetImageDescriptionCTable` function returns the color table for the image described by the image description structure that is referred to by the `desc` parameter. The function correctly sizes the handle for the color table it returns.

DESCRIPTION

The Image Compression Manager stores the custom color table for an image in the appropriate image description structure. Your application must use QuickDraw's `DisposeCTable` routine to free the color table. (For details on `DisposeCTable`, see *Inside Macintosh: Imaging*.)

SPECIAL CONSIDERATIONS

If you want to find out if there is a custom color table, you should check the size of the `CTSize` or `CTSeed` fields in the returned `ctable` parameter. If `CTSize` is 0 or if the `CTSeed` field is less than 0, then the color table is not a custom color table for that image.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memFullErr`	–108	Not enough memory available
`noCodecErr`	–8961	The Image Compression Manager could not find the specified compressor

Working With Pictures and PICT Files

This section describes the functions that let your application compress and decompress single-frame images stored as pictures and PICT files. See "Working With Images," which begins on page 3-73, for information on compressing and manipulating single-frame images stored as pixel map structures. See "Working With Sequences," which begins on page 3-106, for information on compressing and decompressing sequences of images.

As with image compression, the Image Compression Manager provides two sets of functions for working with compressed pictures. If you do not need to control the compression parameters, use the `CompressPicture` or `CompressPictureFile` functions. If you need more control over the operation, use the `FCompressPicture` or `FCompressPictureFile` functions.

The Image Compression Manager automatically expands compressed pictures when you display them. Use the `DrawPictureFile` function to display the contents of a picture file. If you want to alter the spatial characteristics of the image, use the `DrawTrimmedPicture` or `DrawTrimmedPictureFile` functions.

You can work with an image's control information by calling the `GetPictureFileHeader` function.

CompressPicture

The `CompressPicture` function compresses a single-frame image stored as a picture structure and places the result in another picture. If a picture with multiple pixel maps and other graphical objects is passed, the pixel maps will be compressed individually and the other graphic objects will not be affected.

```
pascal OSErr CompressPicture (PicHandle srcPicture,
                              PicHandle dstPicture,
                              CodecQ quality, CodecType cType);
```

`srcPicture`
Contains a handle to the source image, stored as a picture.

`dstPicture`
Contains a handle to the destination for the compressed image. The compressor resizes this handle for the result data.

`quality` Specifies the desired compressed image quality. See "Compression Quality Constants" beginning on page 3-57 for valid values.

`cType` Specifies a compressor type. You must set this parameter to a valid compressor type (see Table 3-3 on page 3-64 for a list of the available compressor types). If the value passed in is 0, or `'raw '`, and the source picture is compressed, the destination picture is created as an uncompressed picture and does not require QuickTime to be displayed.

DESCRIPTION

The `CompressPicture` function compresses only image data. Any other types of data in the picture, such as text, graphics primitives, and previously compressed images, are not modified in any way and are passed through to the destination picture.

This function does not use the graphics port.

If your picture does not fit in memory, use the `CompressPictureFile` function, which is described on page 3-93.

This function supports parameters governing image quality and compressor type. The compressor infers the other compression parameters from the image data in the source picture.

SPECIAL CONSIDERATIONS

The `CompressPicture` function doesn't compress pictures that contain compressed data. Do not alter data in pictures that are already compressed. Instead use `FCompressPicture`, described in the next section.

RESULT CODES

noErr	0	No error
paramErr	–50	Invalid parameter specified
memFullErr	–108	Not enough memory available
noCodecErr	–8961	The Image Compression Manager could not find the specified compressor

SEE ALSO

If you need more control over the compression operation than is provided by the `CompressPicture` function, use the `FCompressPicture` function.

FCompressPicture

The `FCompressPicture` function compresses a single-frame image stored as a picture structure and places the result in another picture. If a picture with multiple pixel maps and other graphical objects is passed, the pixel maps will be compressed individually and the other graphic objects will not be affected.

```
pascal OSErr FCompressPicture (PicHandle srcPicture,
                               PicHandle dstPicture,
                               short colorDepth,
                               CTabHandle clut,
                               CodecQ quality,
                               short doDither,
                               short compressAgain,
                               ProgressProcRecordPtr progressProc,
                               CodecType cType,
                               CompressorComponent codec);
```

`srcPicture`
Contains a handle to the source image, stored as a picture.

`dstPicture`
Contains a handle to the destination for the compressed image. The compressor resizes this handle for the result data.

`colorDepth`
Specifies the depth at which the image is to be compressed. If you set this parameter to 0, the Image Compression Manager determines the appropriate value for the source image. Values of 1, 2, 4, 8, 16, 24, and 32 indicate the number of bits per pixel for color images. Values of 34, 36, and 40 indicate 2-bit, 4-bit, and 8-bit grayscale, respectively, for grayscale images. Your program can determine which depths are supported by a given compressor by examining the compressor information structure

returned by the `GetCodecInfo` function (see "Getting Information About Compressor Components," which begins on page 3-62, for more information).

`clut` Contains a handle to a custom color lookup table. Your program may use this parameter to indicate a custom color lookup table to be used with this image. If the value of the `colorDepth` parameter is less than or equal to 8 and the custom color lookup table is different from that of the source pixel map (that is, the `ctSeed` field values differ in the two pixel maps), the compressor remaps the colors of the image to the custom colors. If you set the `colorDepth` parameter to 16, 24, or 32, the compressor stores the custom color table with the compressed image. The compressor may use the table to specify the best colors to use when displaying the image at lower bit depths. The compressor ignores the `clut` parameter when `colorDepth` is set to 33, 34, 36, or 40. If you set this parameter to `nil`, the compressor uses the color lookup table from the source pixel map.

`quality` Specifies the desired compressed image quality. See "Compression Quality Constants" beginning on page 3-57 for available values.

`doDither` Indicates whether to dither the image. Use this parameter to indicate whether you want the image to be dithered when it is displayed on a lower-resolution screen. The following constants are available:

`defaultDither`
Indicates that the dithering in the source file is to be respected.

`forceDither`
Indicates that the specified image should be dithered whether the source uses dithering or not.

`suppressDither`
Indicates that dithering should not be used in any case. The ability to suppress dithering can be useful if, for example, you have a 32-bit color JPEG image drawn into a 8-bit buffer with a custom color table from the image. In that case, dithering would not be necessary and probably not desirable, particularly if the buffer were to be compressed with the graphics compressor.

`compressAgain`
Indicates whether to recompress compressed image data in the picture. Use this parameter to control whether any compressed image data that is in the source picture should be decompressed and then recompressed using the current parameters. Set the value of this parameter to `true` to recompress such data. Set the value of the parameter to `false` to leave the data as it is. Note that recompressing the data may have undesirable side effects, including image quality degradation.

`progressProc`
Points to a progress function structure. During the compression operation, the compressor may occasionally call a function you provide in order to report its progress (see "Progress Functions" beginning on

page 3-152 for more information about progress functions). If you have not provided a progress function, set this parameter to `nil`. If you pass a value of –1, you obtain a standard progress function.

`cType` Specifies a compressor type. You must set this parameter to a valid compressor type (see Table 3-3 on page 3-64 for a list of the available compressor types). If the value passed in is 0, or `'raw '`, the resulting picture is not compressed and does not require QuickTime to be displayed.

`codec` Contains a compressor identifier. Specify a particular compressor by setting this parameter to its compressor identifier. Alternatively, you may use one of the special identifiers:

`anyCodec` Choose the first compressor of the specified type

`bestSpeedCodec` Choose the fastest compressor of the specified type

`bestFidelityCodec` Choose the most accurate compressor of the specified type

`bestCompressionCodec` Choose the compressor that produces the smallest resulting data

You can also specify a component instance. This may be useful if you have previously set some parameter on a specific instance of a `codec` field and want to make sure that the specified instance is used for that operation.

DESCRIPTION

The `FCompressPicture` function compresses only image data. Any other types of data in the picture, such as text, graphics primitives, and previously compressed images, are not modified in any way and are passed through to the destination picture.

This function supports parameters governing image quality, compressor type, image depth, custom color tables, and dithering.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memFullErr`	–108	Not enough memory available
`noCodecErr`	–8961	The Image Compression Manager could not find the specified compressor
`codecSpoolErr`	–8966	Error loading or unloading data
`codecAbortErr`	–8967	Operation aborted by the progress function

SEE ALSO

If you do not need such a high degree of control over the compression operation, use the `CompressPicture` function, described on page 3-89.

CompressPictureFile

The `CompressPictureFile` function compresses a single-frame image stored as a picture file (PICT file) and places the result in another picture file.

```
pascal OSErr CompressPictureFile (short srcRefNum,
                                  short dstRefNum,
                                  CodecQ quality,
                                  CodecType cType);
```

`srcRefNum` Contains a file reference number for the source PICT file.

`dstRefNum` Contains a file reference number for the destination PICT file. Note that the compressor overwrites the contents of the file referred to by `dstRefNum`. You must open this file with write permission. The destination file can be the same as the source file specified by the `srcRefNum` parameter.

`quality` Specifies the desired compressed image quality. See “Compression Quality Constants” on page 3-57 for available values.

`cType` Specifies a compressor type. You must set this parameter to a valid compressor type (see Table 3-3 on page 3-64 for a list of the available compressor types). If the value passed in is 0, or `'raw '`, and the source picture is compressed, the destination picture is created as an uncompressed picture and does not require QuickTime to be displayed.

DESCRIPTION

The `CompressPictureFile` function compresses only image data. Any other types of data in the file, such as text, graphics primitives, and previously compressed images, are not modified in any way and are passed through to the destination picture. This function does not use the graphics port.

This function supports parameters governing image quality and compressor type. The compressor infers the other compression parameters from the image data in the source picture file.

SPECIAL CONSIDERATIONS

The `CompressPictureFile` function doesn’t compress pictures that contain compressed data. Do not alter data in pictures that are already compressed. Instead use `FCompressPictureFile`, described in the next section.

RESULT CODES

noErr	0	No error
paramErr	–50	Invalid parameter specified
memFullErr	–108	Not enough memory available
noCodecErr	–8961	The Image Compression Manager could not find the specified compressor

File Manager errors

SEE ALSO

If you need more control over the compression operation, use the `FCompressPictureFile` function.

FCompressPictureFile

The `FCompressPictureFile` function compresses a single-frame image stored as a picture file (PICT file) and places the result in another picture file.

```
pascal OSErr FCompressPictureFile (short srcRefNum,
                                   short dstRefNum, short colorDepth,
                                   CTabHandle clut, CodecQ quality,
                                   short doDither,
                                   short compressAgain,
                                   ProgressProcRecordPtr progressProc,
                                   CodecType cType,
                                   CompressorComponent codec);
```

`srcRefNum` Specifies a file reference number for the source PICT file.

`dstRefNum` Specifies a file reference number for the destination PICT file. Note that the compressor overwrites the contents of the file referred to by `dstRefNum`. You must open this file with write permissions. The destination file may be the same as the source file specified by the `srcRefNum` parameter.

`colorDepth`

Specifies the depth at which the image is to be compressed. If you set this parameter to 0, the Image Compression Manager determines the appropriate value for the source image. Values of 1, 2, 4, 8, 16, 24, and 32 indicate the number of bits per pixel for color images. Values of 34, 36, and 40 indicate 2-bit, 4-bit, and 8-bit grayscale, respectively, for grayscale images. Your program can determine which depths are supported by a given compressor by examining the compressor capability structure returned by the `GetCodecInfo` function (see “Getting Information About Compressor Components,” which begins on page 3-62, for more information).

`clut` Contains a handle to a custom color lookup table. Your program may use this parameter to indicate a custom color lookup table to be used with this image. If the value of the `colorDepth` parameter is less than or equal to 8 and the custom color lookup table is different from that of the source pixel map (that is, the `ctSeed` field values differ in the two pixel maps), the compressor remaps the colors of the image to the custom colors. If you set the `colorDepth` parameter to 16, 24, or 32, the compressor stores the custom color table with the compressed image. The compressor may use the table to specify the best colors to use when displaying the image at lower bit depths. The compressor ignores the `clut` parameter when `colorDepth` is set to 33, 34, 36, or 40. If you set this parameter to `nil`, the compressor uses the color lookup table from the source pixel map.

`quality` Specifies the desired compressed image quality. See "Compression Quality Constants" on page 3-57 for available values.

`doDither` Indicates whether to dither the image. Use this parameter to indicate whether you want the image to be dithered when it is displayed on a lower-resolution screen. The following constants are available:

`defaultDither`
: Indicates that the dithering in the source file is to be respected.

`forceDither`
: Indicates that the specified image should be dithered whether the source uses dithering or not.

`suppressDither`
: Indicates that dithering should not be used in any case. The ability to suppress dithering can be useful if, for example, you have a 32-bit, color JPEG image drawn into an 8-bit buffer with a custom color table from the image. In that case, dithering would not be necessary and probably not desirable, particularly if the buffer were to be compressed with the graphics compressor.

`compressAgain`
: Indicates whether to recompress compressed image data in the picture. Use this parameter to control whether any compressed image data that is in the source picture should be decompressed and then recompressed using the current parameters. Set the value of this parameter to `true` to recompress such data. Set the value of this parameter to `false` to leave the data as it is. Note that recompressing the data may have undesirable side effects, including image quality degradation.

`progressProc`
: Points to a progress function structure. During the compression operation, the compressor may occasionally call a function you provide in order to report its progress (see "Progress Functions" on page 3-152 for more information about progress functions). If you have not provided a progress function, set this parameter to `nil`. If you pass a value of –1, you obtain a standard progress function.

`cType` Specifies a compressor type. You must set this parameter to a valid compressor type (see Table 3-3 on page 3-64 for a list of the available compressor types). If the value passed in is 0, or `'raw '` and the source picture is compressed, the destination picture is created as an uncompressed picture and does not require QuickTime to be displayed.

`codec` Contains a compressor identifier. Specify a particular compressor by setting this parameter to its compressor identifier. Alternatively, you may use one of the special identifiers:

`anyCodec` Choose the first compressor of the specified type

`bestSpeedCodec` Choose the fastest compressor of the specified type

`bestFidelityCodec` Choose the most accurate compressor of the specified type

`bestCompressionCodec` Choose the compressor that produces the smallest resulting data

You can also specify a component instance. This may be useful if you have previously set some parameter on a specific instance of a `codec` field and want to make sure that the specified instance is used for that operation.

DESCRIPTION

The `FCompressPicture` function compresses only image data. Any other types of data in the file, such as text, graphics primitives, and previously compressed images, are not modified in any way and are passed through to the destination picture file.

This function supports parameters governing image quality, compressor type, image depth, custom color tables, and dithering.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memFullErr`	–108	Not enough memory available
`noCodecErr`	–8961	The Image Compression Manager could not find the specified compressor
`codecAbortErr`	–8967	Operation aborted by the progress function

File Manager errors

SEE ALSO

If you do not need such a high degree of control over the compression operation, use the `CompressPictureFile` function, described on page 3-93.

DrawPictureFile

The `DrawPictureFile` function draws an image from a specified picture file (PICT file) in the current graphics port. Your program also specifies the destination rectangle for the image.

```
pascal OSErr DrawPictureFile (short refNum, const Rect *frame,
                              ProgressProcRecordPtr progressProc);
```

`refNum` Contains a file reference number for the source PICT file.

`frame` Contains a pointer to the rectangle into which the image is to be loaded. The compressor scales the source image to fit into this destination rectangle.

`progressProc`
Points to a progress function structure. During the operation, the draw function may occasionally call a function you provide in order to report its progress (see "Progress Functions" beginning on page 3-152 for more information about progress functions). If you have not provided a progress function, set this parameter to `nil`. If you pass a value of –1, you obtain a standard progress function.

DESCRIPTION

The `DrawPictureFile` function is essentially the same as QuickDraw's `DrawPicture` routine, except that `DrawPictureFile` reads the picture from disk. (For details on `DrawPicture`, see *Inside Macintosh: Imaging*.) The Image Compression Manager performs any spooling that may be necessary when reading the picture file.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memFullErr`	–108	Not enough memory available
`codecAbortErr`	–8967	Operation aborted by the progress function

File Manager errors

DrawTrimmedPicture

The `DrawTrimmedPicture` function draws an image that is stored as a picture into the current graphics port and trims that image to fit a region you specify.

```
pascal OSErr DrawTrimmedPicture (PicHandle srcPicture,
                                 const Rect *frame, RgnHandle trimMask,
                                 short doDither,
                                 ProgressProcRecordPtr progressProc);
```

`srcPicture` Contains a handle to the source image; stored as a picture.

`frame` Contains a pointer to the rectangle into which the decompressed image is to be loaded.

`trimMask` Contains a handle to a clipping region in the destination coordinate system. The decompressor applies this mask to the destination image and ignores any image data that fall outside the specified region. Set this parameter to `nil` if you do not want to clip the source image. In this case, this function acts like QuickDraw's `DrawPicture` routine, but it also allows you to control dithering and assign a progress function. (See *Inside Macintosh: Imaging* for more on `DrawPicture`.)

`doDither` Indicates whether to dither the image. Use this parameter to indicate whether you want the image to be dithered when it is displayed on a lower-resolution screen. The following constants are available:

`defaultDither`
Indicates that the dithering in the source file is to be respected.

`forceDither`
Indicates that the specified image should be dithered whether the source uses dithering or not.

`suppressDither`
Indicates that dithering should not be used in any case. The ability to suppress dithering can be useful if, for example, you have a 32-bit, color JPEG image drawn into an 8-bit buffer with a custom color table from the image. In that case, dithering would not be necessary and probably not desirable, particularly if the buffer were to be compressed with the graphics compressor.

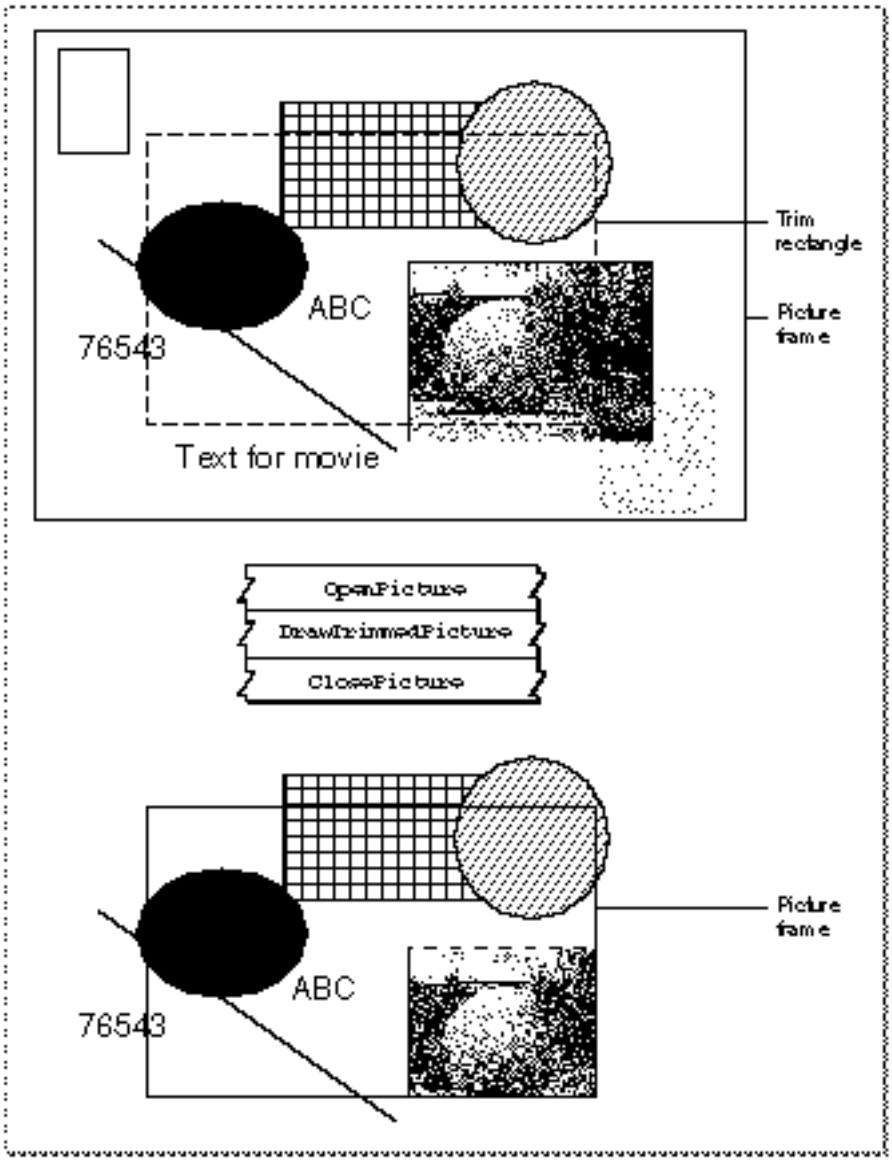
`progressProc`
Points to a progress function structure. During the compression operation, the compressor may occasionally call a function you provide in order to report its progress (see "Progress Functions" beginning on page 3-152 for more information about progress functions). If you have not provided a progress function, set this parameter to `nil`. If you pass a value of –1, you obtain a standard progress function.

DESCRIPTION

The `DrawTrimmedPicture` function works with compressed image data—the source data stays compressed. The function trims the image to fit the specified clipping region. Figure 3-10 shows how the `DrawTrimmedPicture` function works. It illustrates how you can use this function to save part of a picture (the clipped or uncompressed image data that is not within the trim region is ignored and is not included in the destination picture). All the remaining objects in the resulting image are clipped. You use QuickDraw's `OpenPicture` and `ClosePicture` routines to open and close the destination picture. (For more on `OpenPicture` and `ClosePicture`, see *Inside Macintosh: Imaging*.)

Note that if you just use a clip while making a picture, the data—though not visible—is still stored in the picture.

Figure 3-10 The operation of the `DrawTrimmedPicture` function



RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memFullErr`	–108	Not enough memory available
`codecAbortErr`	–8967	Operation aborted by the progress function

SEE ALSO

If your source image does not fit in memory, use the `DrawTrimmedPictureFile` function, which is described in the next section.

DrawTrimmedPictureFile

The `DrawTrimmedPictureFile` function draws an image that is stored as a picture file (PICT file) into the current graphics port and trims that image to fit a region you specify.

```
pascal OSErr DrawTrimmedPictureFile (short srcRefnum,
                                     const Rect *frame,
                                     RgnHandle trimMask,
                                     short doDither,
                                     ProgressProcRecordPtr progressProc);
```

`srcRefNum` Contains a file reference number for the source PICT file.

`frame` Contains a pointer to the rectangle into which the decompressed image is to be loaded.

`trimMask` Contains a handle to a clipping region in the destination coordinate system. The decompressor applies this mask to the destination image and ignores any image data that fall outside the specified region. Set this parameter to `nil` if you do not want to clip the source image. In this case, this function acts like the `DrawPictureFile` function, which is described on page 3-97.

`doDither` Indicates whether to dither the image. Use this parameter to indicate whether you want the image to be dithered when it is displayed on a lower-resolution screen. The following constants are available:

`defaultDither`
Indicates that the dithering in the source picture file is to be respected.

`forceDither`
Indicates that the specified image should be dithered whether the source uses dithering or not.

`suppressDither`
Indicates that dithering should not be used in any case. The ability to suppress dithering can be useful if, for example, you have a 32-bit color JPEG image drawn into an 8-bit buffer with a custom color table from the image. In that case, dithering would not be necessary and probably not desirable, particularly if the buffer were to be compressed with the graphics compressor.

`progressProc`
Points to a progress function structure. During the compression operation, the compressor may occasionally call a function you provide in order to report its progress (see “Progress Functions” beginning on page 3-152 for more information about progress functions). If you have not provided a progress function, set this parameter to `nil`. If you pass a value of –1, you obtain a standard progress function.

DESCRIPTION

The `DrawTrimmedPictureFile` function works with compressed image data—the source data stays compressed. The function trims the image to fit the specified clipping region. The Image Compression Manager handles any spooling issues that may arise when reading the picture file.

You can use this function to save part of a picture, since the image data that is not within the trim region is ignored and is not included in the destination picture file. All the reamining objects in the resulting object are clipped.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memFullErr`	–108	Not enough memory available
`codecAbortErr`	–8967	Operation aborted by the progress function

File Manager errors

GetPictureFileHeader

The `GetPictureFileHeader` function extracts the picture frame (the `picFrame` rectangle in the picture structure) and file header from a specified picture file (PICT file). Your program can use this information to determine how to draw an image without having to read the picture file.

```
pascal OSErr GetPictureFileHeader (short refNum, Rect *frame,
                                   OpenCPicParams *header);
```

`refNum` Contains a file reference number for the source PICT file.

`frame` Contains a pointer to a rectangle that is to receive the picture frame rectangle of the picture file. This function places the `picFrame` rectangle from the picture structure into the rectangle referred to by the `frame` parameter. If you are not interested in this information, pass `nil` in this parameter.

`header` Contains a pointer to an `OpenCPicture` parameters structure. The `GetPictureFileHeader` function places the header from the specified picture file into the structure referred to by the `header` parameter. Note that this function always returns a version 2 header. If the source file is a version 1 PICT file, the `GetPictureFileHeader` function converts the header into version 2 format before returning it to your application. See *Inside Macintosh: Imaging* for more information about picture headers and the `OpenCPicture` function. If you are not interested in this information, pass `nil` in this parameter.

DESCRIPTION

The `GetPictureFileHeader` function always returns a version 2 PICT file header. If the specified picture file is a version 1 file, the `GetPictureFileHeader` function synthesizes a version 2 header from the information available in the file header.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified

File Manager errors

Making Thumbnail Pictures

This section describes the functions that allow your application to create thumbnail pictures from existing images that are stored as pixel maps, pictures, or picture files. Thumbnail pictures are useful for creating small, representative images of a source image. You can use thumbnails when you create previews for files that contain image data (for more information about file previews, see the chapter "Movie Toolbox" in this book).

You can create thumbnails from pictures, picture files, or pixel maps—use the `MakeThumbnailFromPicture`, `MakeThumbnailFromPictureFile`, or `MakeThumbnailFromPixMap` function, as appropriate.

MakeThumbnailFromPicture

The `MakeThumbnailFromPicture` function creates an 80-by-80 pixel thumbnail picture from a specified picture structure.

```
pascal OSErr MakeThumbnailFromPicture (PicHandle picture,
                                       short colorDepth,
                                       PicHandle thumbnail,
                                       ProgressProcRecordPtr progressProc);
```

`picture` Contains a handle to the image from which the thumbnail is to be extracted. The image must be stored in a picture structure.

`colorDepth`
Specifies the depth at which the image is likely to be viewed. If you set this parameter to 0, the Image Compression Manager determines the appropriate value for the source image. Values of 1, 2, 4, 8, 16, 24, and 32 indicate the number of bits per pixel for color images. Values of 34, 36, and 40 indicate 2-bit, 4-bit, and 8-bit grayscale, respectively, for grayscale images.

`thumbnail` Contains a handle to the destination picture structure for the thumbnail image. The compressor resizes this handle for the resulting data.

progressProc
Points to a progress function structure. During the operation, the Image Compression Manager will occasionally call a function you provide in order to report its progress (see "Progress Functions" beginning on page 3-152 for more information about progress functions). If you have not provided a progress function, set this parameter to nil. If you pass a value of –1, you obtain a standard progress function.

RESULT CODES

noErr	0	No error
paramErr	–50	Invalid parameter specified
memFullErr	–108	Not enough memory available
codecAbortErr	–8967	Operation aborted by the progress function

MakeThumbnailFromPictureFile

The MakeThumbnailFromPictureFile function creates an 80-by-80 pixel thumbnail picture from a specified picture file (PICT file).

```
pascal OSErr MakeThumbnailFromPictureFile (short refNum,
                                short colorDepth,
                                PicHandle thumbnail,
                                ProgressProcRecordPtr progressProc);
```

refNum Contains a file reference number for the PICT file from which the thumbnail is to be extracted.

colorDepth
Specifies the depth at which the image is likely to be viewed. If you set this parameter to 0, the Image Compression Manager determines the appropriate value for the source image. Values of 1, 2, 4, 8, 16, 24, and 32 indicate the number of bits per pixel for color images. Values of 34, 36, and 40 indicate 2-bit, 4-bit, and 8-bit grayscale, respectively, for grayscale images.

thumbnail Contains a handle to the destination picture structure for the thumbnail image. The compressor resizes this handle for the resulting data.

progressProc
Points to a progress function structure. During the operation, the Image Compression Manager will occasionally call a function you provide in order to report its progress (see "Progress Functions" beginning on page 3-152 for more information about progress functions). If you have not provided a progress function, set this parameter to nil. If you pass a value of –1, you obtain a standard progress function.

RESULT CODES

noErr	0	No error
paramErr	–50	Invalid parameter specified
memFullErr	–108	Not enough memory available
codecAbortErr	–8967	Operation aborted by the progress function

File Manager errors

MakeThumbnailFromPixMap

The MakeThumbnailFromPixMap function creates an 80-by-80 pixel thumbnail picture from a specified pixel map structure.

```
pascal OSErr MakeThumbnailFromPixMap (PixMapHandle src,
                                      const Rect *srcRect,
                                      short colorDepth,
                                      PicHandle thumbnail,
                                      ProgressProcRecordPtr progressProc);
```

src Contains a handle to the image from which the thumbnail is to be extracted. The image must be stored in a pixel map structure.

srcRect Contains a pointer to a rectangle defining the portion of the image to use for the thumbnail.

colorDepth
Specifies the depth at which the image is likely to be viewed. If you set this parameter to 0, the Image Compression Manager determines the appropriate value for the source image. Values of 1, 2, 4, 8, 16, 24, and 32 indicate the number of bits per pixel for color images. Values of 34, 36, and 40 indicate 2-bit, 4-bit, and 8-bit grayscale, respectively, for grayscale images.

thumbnail Contains a handle to the destination picture structure for the thumbnail image. The compressor resizes this handle for the resulting data.

progressProc
Points to a progress function structure. During the operation, the Image Compression Manager will occasionally call a function you provide in order to report its progress (see “Progress Functions” beginning on page 3-152, for more information about progress functions). This parameter contains a pointer to a structure that identifies that progress function. If you have not provided a progress function, set this parameter to nil. If you pass a value of –1, you obtain a standard progress function.

DESCRIPTION

The thumbnail returned is an 80-by-80 pixel picture, but the aspect ratio is maintained.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memFullErr`	–108	Not enough memory available
`codecAbortErr`	–8967	Operation aborted by the progress function

Working With Sequences

This section describes the functions that enable your application to compress and decompress sequences of images. Each image in the sequence is referred to as a *frame*. Note that the sequence carries no time information. The Movie Toolbox manages all temporal aspects of displaying the sequence. Consequently, your application can focus on the order of images in the sequence.

To process a sequence of frames, your program first begins the sequence (by issuing either the `CompressSequenceBegin` or `DecompressSequenceBegin` functions). You then process each frame in the sequence (use `CompressSequenceFrame` to compress a frame; use `DecompressSequenceFrame` to decompress a frame). When you are done, close the sequence by issuing the `CDSequenceEnd` function. You can check on the status of the current operation by calling the `CDSequenceBusy` function.

Note that the Image Compression Manager provides a rich set of functions that allow your application to control many of the parameters that govern sequence processing. You set default values for most of these parameters when you start the sequence. These additional functions allow you to modify those parameters while you are processing a sequence. See “Changing Sequence-Compression Parameters,” which begins on page 3-120, for information on functions that affect sequence compression. See “Changing Sequence-Decompression Parameters” beginning on page 3-129 for information on functions that affect sequence decompression.

CompressSequenceBegin

Your application calls the `CompressSequenceBegin` function to signal the beginning of the process of compressing a sequence of frames. The Image Compression Manager prepares for the sequence-compression operation by reserving appropriate system resources. You must call this function before calling the `CompressSequenceFrame` function, which is described in the next section.

```
pascal OSErr CompressSequenceBegin (ImageSequence *seqID,
                                    PixMapHandle src,
                                    PixMapHandle prev,
                                    const Rect *srcRect,
                                    const  Rect *prevRect,
                                    short colorDepth,
                                    CodecType cType,
                                    CompressorComponent codec,
                                    CodecQ spatialQuality,
                                    CodecQ temporalQuality,
                                    long keyFrameRate,
                                    CTabHandle clut,
                                    CodecFlags flags,
                                    ImageDescriptionHandle desc);
```

`seqID` Contains a pointer to a field to receive the unique identifier for this sequence. You must supply this identifier to all subsequent Image Compression Manager functions that relate to this sequence.

`src` Contains a handle to a pixel map that will contain the image to be compressed. The image must be stored in a pixel map structure.

`prev` Contains a handle to a pixel map that will contain a previous image. The compressor uses this buffer to store a previous image against which the current image (stored in the pixel map referred to by the `src` parameter) is compared when performing temporal compression. This pixel map must be created at the same depth and with the same color table as the source image. The compressor manages the contents of this pixel map based upon several considerations, such as the key frame rate and the degree of difference between compared images. If you want the compressor to allocate this pixel map or if you do not want to perform temporal compression (that is, you have set the value of the `temporalQuality` parameter to 0), set this parameter to `nil`.

You can set or change the previous image buffer for an active sequence by calling the `SetCSequencePrev` function. You can obtain a pointer to a pixel map that was allocated by the compressor by calling the `GetCSequencePrevBuffer` function. See “Changing Sequence-Compression Parameters,” which begins on page 3-120, for information about these functions.

`srcRect` Contains a pointer to a rectangle defining the portion of the image to compress. The compressor applies this rectangle to the image stored in the buffer referred to by the `src` parameter.

`prevRect` Contains a pointer to a rectangle defining the portion of the previous image to use for temporal compression. The compressor uses this portion of the previous image as the basis of comparison with the current image. The compressor ignores this parameter if you have not provided a buffer for previous images. This rectangle must be the same size as the source rectangle, which is specified with the `srcRect` parameter.

You can set or change the rectangle used with the previous image buffer for an active sequence by calling the `SetCSequencePrev` function. See "Changing Sequence-Compression Parameters," which begins on page 3-120, for information about this function.

`colorDepth`

Specifies the depth at which the sequence is likely to be viewed. Compressors may use this as an indication of the color or grayscale resolution of the compressed images. If you set this parameter to 0, the Image Compression Manager determines the appropriate value for the source image. Values of 1, 2, 4, 8, 16, 24, and 32 indicate the number of bits per pixel for color images. Values of 34, 36, and 40 indicate 2-bit, 4-bit, and 8-bit grayscale, respectively, for grayscale images. Your program can determine which depths are supported by a given compressor by examining the compressor information structure returned by the `GetCodecInfo` function (described on page 3-65).

`cType` Specifies a compressor type. You must set this parameter to a valid compressor type (see Table 3-3 on page 3-64 for a list of the available compressor types).

`codec` Specifies a compressor identifier. Specify a particular compressor by setting this parameter to its compressor identifier. Alternatively, you may use one of the special identifiers:

`anyCodec` Choose the first compressor of the specified type

`bestSpeedCodec`
Choose the fastest compressor of the specified type

`bestFidelityCodec`
Choose the most accurate compressor of the specified type

`bestCompressionCodec`
Choose the compressor that produces the smallest resulting data

You can also specify a component instance. This may be useful if you have previously set some parameter on a specific instance of a `codec` field and want to make sure that the specified instance is used for that operation.

If you set the `codec` parameter to `anyCodec`, the Image Compression Manager chooses the first compressor it finds of the specified type.

`spatialQuality`
: Specifies the desired compressed image quality. See “Compression Quality Constants” beginning on page 3-57 for available values. You can change the value of this parameter for an active sequence by calling the `SetCSequenceQuality` function (described on page 3-120).

`temporalQuality`
: Specifies the desired sequence temporal quality. This parameter governs the level of compression you desire with respect to information between successive frames in the sequence. Set this parameter to 0 to prevent the compressor from applying temporal compression to the sequence. See “Compression Quality Constants” beginning on page 3-57 for other available values.

  You can change the value of this parameter for an active sequence by calling the `SetCSequenceQuality` function (described on page 3-120).

`keyFrameRate`
: Specifies the maximum number of frames allowed between key frames. Key frames provide points from which a temporally compressed sequence may be decompressed. Use this parameter to control the frequency at which the compressor places key frames into the compressed sequence. The compressor determines the optimum placement for key frames based upon the amount of redundancy between adjacent images in the sequence. Consequently, the compressor may insert key frames more frequently than you have requested. However, the compressor never places fewer key frames than is indicated by the setting of the `keyFrameRate` parameter. The compressor ignores this parameter if you have not requested temporal compression (that is, you have set the `temporalQuality` parameter to 0). If you pass in 0 in this parameter, this indicates that there are no key frames in the sequence. If you pass in any other number, it specifies the number of non-key frames between key frames. Set this parameter to 1 to specify all key frames, to 2 to specify every other frame as a key frame, to 3 to specify every third frame as a key frame, and so forth.

  Your application may change the key frame rate for an active sequence by calling the `SetCSequenceKeyFrameRate` function (described beginning on page 3-121). See “Defining Key Frame Rates” on page 3-47 for more information about key frames.

`clut`
: Contains a handle to a custom color lookup table. Your program may use this parameter to indicate a custom color lookup table to be used with this image. If the value of the `colorDepth` parameter is less than or equal to 8 and the custom color lookup table is different from that of the source pixel map (that is, the `ctSeed` field values differ in the two pixel maps), the compressor remaps the colors of the image to the custom colors. If you set the `colorDepth` parameter to 16, 24, or 32, the compressor stores the custom color table with the compressed image. The compressor may use the table to specify the best colors to use when displaying the image at lower bit depths. The compressor ignores the `clut` parameter when `colorDepth` is set to 33, 34, 36, or 40. If you set this parameter to `nil`, the compressor uses the color lookup table from the source pixel map.

`flags` Contains flags providing further control information. See "Image Compression Manager Function Control Flags," which begins on page 3-58, for information about `CodecFlags` fields. You must set either the value of the `codecFlagUpdatePrevious` flag or the `codecFlagUpdatePreviousComp` flag to 1 (be sure to set unused flags to 0). The following flags are available for this function:

`codecFlagUpdatePrevious`
Controls whether the compressor updates the previous image during compression. This flag is only used with sequences that are being temporally compressed. If you set this flag to 1, the compressor copies the current frame into the previous frame buffer at the end of frame compression.

`codecFlagUpdatePreviousComp`
Controls whether the compressor updates the previous image buffer with the compressed image. This flag is only used with temporal compression and is similar to the `codecFlagUpdatePrevious` flag. As with the `codecFlagUpdatePrevious` flag, if you set this flag to 1, the compressor updates the previous frame buffer at the end of frame compression. However, this flag causes the Image Compression Manager to update the frame buffer using an image obtained by decompressing the results of the most recent compression operation, rather than the source image, which may give better results at the expense of taking more time.

`codecFlagWasCompressed`
Indicates to the compressor that the image to be compressed has been compressed before. This information may be useful to compressors that can compensate for the image degradation that may otherwise result from repeated compression and decompression of the same image. Set this flag to 1 to indicate that the image was previously compressed. Set this flag to 0 if the image was not previously compressed.

`desc` Contains a handle that is to receive a formatted image description structure. The Image Compression Manager resizes this handle for the returned image description structure. Your application should store this image description with the compressed sequence. During the compression operation, the Image Compression Manager and the compressor component update the contents of this image description. Consequently, you should not store the image description until the sequence has been completely processed.

RESULT CODES

noErr	0	No error
paramErr	–50	Invalid parameter specified
memFullErr	–108	Not enough memory available
noCodecErr	–8961	The Image Compression Manager could not find the specified compressor
codecConditionErr	–8972	Component cannot perform requested operation

CompressSequenceFrame

Your application calls the `CompressSequenceFrame` function to compress one of a sequence of frames.

```
pascal OSErr CompressSequenceFrame (ImageSequence seqID,
                    PixMapHandle src, const Rect *srcRect,
                    CodecFlags flags, Ptr data, long *dataSize,
                    unsigned char *similarity,
                    CompletionProcRecordPtr asyncCompletionProc);
```

`seqID` Unique sequence identifier that was returned by the `CompressSequenceBegin` function (described in the previous section).

`src` Contains a handle to a pixel map that contains the image to be compressed. The image must be stored in a pixel map structure.

`srcRect` Contains a pointer to a rectangle defining the portion of the image to compress. The compressor applies this rectangle to the image stored in the buffer referred to by the `src` parameter.

`flags` Specifies flags providing further control information. See "Image Compression Manager Function Control Flags," which begins on page 3-58, for information about `CodecFlags` fields. You must set the value of either the `codecFlagUpdatePrevious` flag or the `codecFlagUpdatePreviousComp` flag to 1 (be sure to set unused flags to 0). The following flags are available for this function:

`codecFlagUpdatePrevious`

Controls whether the compressor updates the previous image during compression. This flag is only used with sequences that are being temporally compressed. If you set this flag to 1, the compressor copies the current frame into the previous frame buffer at the end of frame compression.

Indicates to the compressor that the image to be compressed has been compressed before. This information may be useful to compressors that can compensate for the image degradation that may otherwise result from repeated compression and decompression of the same image. Set this flag to 1 to indicate that the image was previously compressed. Set this flag to 0 if the image was not previously compressed.

codecFlagUpdatePreviousComp
: Controls whether the compressor updates the previous image buffer with the compressed image. This flag is only used with temporal compression and is similar to the `codecFlagUpdatePrevious` flag. As with the `codecFlagUpdatePrevious` flag, if you set this flag to 1, the compressor updates the previous frame buffer at the end of frame compression. However, this flag causes the Image Compression Manager to update the frame buffer using an image obtained by decompressing the results of the most recent compression operation, rather than the source image.

codecFlagForceKeyFrame
: Controls whether the compressor creates a key frame from the current image. This flag is only used with temporal compression. If you set this flag to 1, the compressor makes the current image a key frame. If you set this flag to 0, the compressor decides based on other criteria, such as the key frame rate, whether to create a key frame from the current image. If you don't want any key frames other than the ones that are forced, set the key frame rate for the sequence to 0.

codecFlagLiveGrab
: Indicates to the compressor that speed is of the utmost importance, and that size and quality are of lesser importance. This flag is useful when you are grabbing sequences from a live source where each frame must be compressed quickly.

`data` Points to a location to receive the compressed image data. It is your program's responsibility to make sure that this location can receive at least as much data as indicated by the `GetMaxCompressionSize` function (described on page 3-68). The Image Compression Manager places the actual size of the compressed image into the field referred to by the `dataSize` parameter. This pointer must contain a 32-bit clean address. If you use a dereferenced, locked handle, you must call the Memory Manager's `StripAddress` routine before you use that pointer with this parameter. For details on `StripAddress`, see *Inside Macintosh: Memory*.

`dataSize` Contains a pointer to a field that is to receive the size, in bytes, of the compressed image.

`similarity` Contains a pointer to a field that is to receive a similarity value. The `CompressSequenceFrame` function returns a value that indicates the similarity of the current frame to the previous frame. A value of 0 indicates that the current frame is a key frame in the sequence. A value of

255 indicates that the current frame is identical to the previous frame. Values from 1 through 254 indicate relative similarity, ranging from very different (1) to very similar (254).

`asyncCompletionProc`
Points to a completion function structure. The compressor calls your completion function when an asynchronous compression operation is complete. You can cause the compression to be performed asynchronously by specifying a completion function if the compressor supports asynchronous compression. For more information about completion function structures, see "Completion Functions" on page 3-154.

If you specify asynchronous operation, you must not read the compressed data until the compressor indicates that the operation is complete by calling your completion function. Set `asyncCompletionProc` to `nil` to specify synchronous compression. If you set `asyncCompletionProc` to –1, the operation is performed asynchronously but the compressor does not call your completion function.

If the `asyncCompletionProc` parameter is not `nil`, the following conditions are in effect: the pixels in the source image must stay valid until the completion function is called with its `codecCompletionSource` flag, and the resulting compressed data is not valid until it is called with its `codecCompletionDest` flag set.

SPECIAL CONSIDERATIONS

You must call the `CompressSequenceBegin` function (described in the previous section) shortly before you use the `CompressSequenceFrame` function. `CompressSequenceFrame` uses the current graphics device and port set from your prior call to `CompressSequenceBegin`.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memFullErr`	–108	Not enough memory available
`noCodecErr`	–8961	The Image Compression Manager could not find the specified compressor
`codecSpoolErr`	–8966	Error loading or unloading data

DecompressSequenceBegin

The Movie Toolbox handles the details of decompressing image sequences in QuickTime movies. If you need to decompress other sequences, your application calls this function to signal the beginning of the process of decompressing a sequence of frames. You must

call this function before calling the DecompressSequenceFrame function (described in the next section).

```
pascal OSErr DecompressSequenceBegin (ImageSequence *seqID,
                                      ImageDescriptionHandle desc,
                                      CGrafPtr port, GDHandle gdh,
                                      const Rect *srcRect,
                                      MatrixRecordPtr matrix, short mode,
                                      RgnHandle mask, CodecFlags flags,
                                      CodecQ accuracy,
                                      DecompressorComponent codec);
```

seqID — Contains a pointer to a field to receive the unique identifier for this sequence returned by the CompressSequenceBegin function (described on page 3-106). You must supply this identifier to all subsequent Image Compression Manager functions that relate to this sequence.

desc — Contains a handle to the image description structure that describes the compressed image.

port — Points to the graphics port for the destination image. If this parameter specifies a graphics world or points to the screen, set the gdh parameter to nil. If you set this parameter to nil, the Image Compression Manager uses the current port (in this case, you should also set the gdh parameter to nil).

gdh — Contains a handle to the graphics device record for the destination image. If the port parameter specifies a graphics world or the screen, or if you set the port parameter to nil, set this parameter to nil.

srcRect — Contains a pointer to a rectangle defining the portion of the image to decompress. This rectangle must lie within the boundary rectangle of the compressed image, which is defined by (0,0) and ((**desc).width,(**desc).height). If you want to decompress the entire source image, set this parameter to nil. If the srcRect parameter is nil, the rectangle is set to the rectangle structure of the image description structure. Your application can change the source rectangle for an active sequence by calling the SetDSequenceSrcRect function (described on page 3-131).

matrix — Points to a matrix structure that specifies how to transform the image during decompression. You can use the matrix structure to translate or scale the image during decompression. If you do not want to apply such effects, set the matrix parameter to nil. For more information about matrix operations, see the chapter “Movie Toolbox” in this book.

Your application can change the matrix for an active sequence by calling the SetDSequenceMatrix function (described on page 3-131).

mode — Specifies the transfer mode for the operation. The Image Compression Manager supports the same transfer modes supported by QuickDraw’s CopyBits routine (described in *Inside Macintosh: Imaging*).

Your application can change the transfer mode for an active sequence by calling the `SetDSequenceTransferMode` function (described on page 3-130).

`mask` Contains a handle to a clipping region in the destination coordinate system. If specified, the decompressor applies this mask to the destination image. If you do not want to mask pixels in the destination, set this parameter to `nil`.

Your application can change the clipping mask for an active sequence by calling the `SetDSequenceMask` function (described on page 3-132).

`flags` Contains flags providing further control information. See "Image Compression Manager Function Control Flags," which begins on page 3-58, for information about `CodecFlags` fields. The following flags are available for this function:

`codecFlagUseScreenBuffer`
Controls whether the decompressor allocates an offscreen buffer. The decompressor places the decompressed image into that buffer and then copies the image to the destination pixel map after completing the decompression operation. Using an offscreen buffer reduces the tearing effect that can result from writing directly to the screen during decompression. Set this flag to 1 to cause the decompressor to allocate and use an offscreen buffer. Set this flag to 0 to cause the decompressor to write to the destination pixel map.

Your application can determine the screen buffer for an active sequence by calling the `GetDSequenceScreenBuffer` function (described on page 3-136).

`codecFlagUseImageBuffer`
Controls whether the decompressor allocates an offscreen buffer for the current image. The decompressor uses this buffer to store the compressed data from the current image so that subsequent images that are temporally compressed can be processed correctly. Set this flag to 1 to cause the decompressor to use an image buffer. Set this flag to 0 if your sequence is not temporally compressed and therefore does not require the use of an image buffer.

Your application can determine the image buffer for an active sequence by calling the `GetDSequenceImageBuffer` function (described on page 3-136).

`accuracy` Specifies the accuracy desired in the decompressed image. Values for this parameter are on the same scale as compression quality. See "Compression Quality Constants" beginning on page 3-57 for available values.

Your application can change the accuracy parameter for an active sequence by calling the `SetDSequenceAccuracy` function (described on page 3-134).

codec Contains a compressor identifier. Specify a particular decompressor by setting this parameter to its compressor identifier. Alternatively, you may use one of the special identifiers:

anyCodec Choose the first decompressor of the specified type

bestSpeedCodec
Choose the fastest decompressor of the specified type

bestFidelityCodec
Choose the most accurate decompressor of the specified type

You can also specify a component instance. This may be useful if you have previously set some parameter on a specific instance of a codec field and want to make sure that the specified instance is used for that operation.

If you set the codec parameter to anycodec, the Image Compression Manager chooses the first decompressor it finds of the specified type.

DESCRIPTION

Use the SetDSequenceDataProc function (described on page 3-135) to assign a data-loading function to the sequence. Use the SetDSequenceMatte function (described on page 3-133) to assign a blend matte to the sequence.

RESULT CODES

noErr	0	No error
paramErr	–50	Invalid parameter specified
memFullErr	–108	Not enough memory available
noCodecErr	–8961	The Image Compression Manager could not find the specified compressor
codecScreenBufErr	–8964	Could not allocate the screen buffer
codecImageBufErr	–8965	Could not allocate the image buffer
codecConditionErr	–8972	Component cannot perform requested operation

DecompressSequenceFrame

Your application calls the DecompressSequenceFrame function to decompress one of a sequence of frames. You must have called the DecompressSequenceBegin function before calling this function. You specify the destination with the port parameter to the DecompressSequenceBegin function, described in the previous section.

```
pascal OSErr DecompressSequenceFrame (ImageSequence seqID,
              Ptr data, CodecFlags inFlags,
              CodecFlags *outFlags,
              CompletionProcRecordPtr asyncCompletionProc);
```

`seqID` Contains the unique sequence identifier that was returned by the `DecompressSequenceBegin` function (described on page 3-113).

`data` Points to the compressed image data. This pointer must contain a 32-bit clean address. If you use a dereferenced, locked handle, you must call the Memory Manager's `StripAddress` routine before you use that pointer with this parameter.

`inFlags` Contains flags providing further control information. See "Image Compression Manager Function Control Flags," which begins on page 3-58, for information about `CodecFlags` fields. The following flags are available for this function:

`codecFlagNoScreenUpdate`
Controls whether the decompressor updates the screen image. If you set this flag to 1, the decompressor does not write the current frame to the screen, but does write the frame to its offscreen image buffer (if one was allocated). If you set this flag to 0, the decompressor writes the frame to the screen.

`codecFlagDontOffscreen`
Controls whether the decompressor uses the offscreen buffer during sequence decompression. This flag is only used with sequences that have been temporally compressed. If this flag is set to 1, the decompressor does not use the offscreen buffer during decompression. Instead, the decompressor returns an error. This allows your application to refill the offscreen buffer. If this flag is set to 0, the decompressor uses the offscreen buffer if appropriate.

`codecFlagOnlyScreenUpdate`
Controls whether the decompressor decompresses the current frame. If you set this flag to 1, the decompressor writes the contents of its offscreen image buffer to the screen, but does decompress the current frame. If you set this flag to 0, the decompressor decompresses the current frame and writes it to the screen. You can set this flag to 1 only if you have allocated an offscreen image buffer for use by the decompressor.

`outFlags` Contains a pointer to status flags. The decompressor updates these flags at the end of the decompression operation. See "Image Compression Manager Function Control Flags," which begins on page 3-58, for information about `CodecFlags` constants. The following flags may be set by this function:

`codecFlagUsedNewImageBuffer`
Indicates to your application that the decompressor used the offscreen image buffer for the first time when it processed this frame. If this flag is set to 1, the decompressor used the image buffer for this frame and this is the first time the decompressor used the image buffer in this sequence.

codecFlagUsedImageBuffer
: Indicates whether the decompressor used the offscreen image buffer. If the decompressor used the image buffer during the decompress operation, it sets this flag to 1. Otherwise, it sets this flag to 0.

codecFlagDontUseNewImageBuffer
: Forces an error to be returned when a new image buffer would have to be allocated instead of allocating the new buffer.

codecFlagInterlaceUpdate
: Updates the screen by **interlacing** even and odd scan lines to reduce **tearing** artifacts (if the decompressor supports this mode).

asyncCompletionProc
: Points to a completion function structure. The compressor calls your completion function when an asynchronous decompression operation is complete. You can cause the decompression to be performed asynchronously by specifying a completion function. See "Completion Functions," which begins on page 3-154, for more information about completion functions.

  If you specify asynchronous operation, you must not read the decompressed image until the decompressor indicates that the operation is complete by calling your completion function. Set `asyncCompletionProc` to `nil` to specify synchronous decompression. If you set `asyncCompletionProc` to –1, the operation is performed asynchronously but the decompressor does not call your completion function.

SPECIAL CONSIDERATIONS

Only if the `asyncCompletionProc` parameter of `CompressSequenceFrame` is not `nil` are the following conditions in effect: the compressed data must remain valid until the completion function is called with its `codecCompletionSource` flag, and the pixels in the destination image will not be valid until the completion function is called with its `codecCompletionDest` flag set.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memFullErr`	–108	Not enough memory available
`noCodecErr`	–8961	The Image Compression Manager could not find the specified compressor
`codecSpoolErr`	–8966	Error loading or unloading data

CDSequenceBusy

Your application may call the `CDSequenceBusy` function to check the status of an asynchronous compression or decompression operation.

```
pascal OSErr CDSequenceBusy (ImageSequence seqID);
```

`seqID` Contains the unique sequence identifier that was returned by the `DecompressSequenceBegin` or `CompressSequenceBegin` function (described on page 3-113 and page 3-106, respectively).

DESCRIPTION

If there is no asynchronous operation in progress, the `CDSequenceBusy` function returns a 0 result code. If there is an asynchronous operation in progress, the result code is 1. Negative result codes indicate an error.

SPECIAL CONSIDERATIONS

If you call the `CDSequenceEnd` function (described in the next section), you don't need to call `CDSequenceBusy` to make sure you have completed an operation.

RESULT CODES

`paramErr`	–50	Invalid parameter specified
`codecUnimpErr`	–8962	Feature not implemented by this compressor

Component Manager errors

CDSequenceEnd

Your application calls the `CDSequenceEnd` function to indicate the end of processing for an image sequence.

```
pascal OSErr CDSequenceEnd (ImageSequence seqID);
```

`seqID` Contains the unique sequence identifier that was returned by the `DecompressSequenceBegin` or `CompressSequenceBegin` function (described on page 3-113 and page 3-106, respectively).

SPECIAL CONSIDERATIONS

You must make this call to `CDSequenceEnd` to make sure that all resources associated with the sequence are freed.

RESULT CODES

noErr	0	No error
paramErr	–50	Invalid parameter specified
noCodecErr	–8961	The Image Compression Manager could not find the specified compressor

SEE ALSO

See "Compressing Sequences," which begins on page 3-31, and "Decompressing Sequences," which begins on page 3-33, for more on how to use CDSequenceEnd. Also see "A Sample Program for Compressing and Decompressing a Sequence of Images," which begins on page 3-35, for details on how to use CDSequenceEnd.

Changing Sequence-Compression Parameters

This section describes the functions that allow your application to manipulate the parameters that control sequence compression and to get information about memory that the compressor has allocated. You can use these functions during the sequence-compression process. Your application establishes the default value for most of these parameters with the CompressSequenceBegin function (described on page 3-106). Some of these functions deal with parameter values that cannot be set when starting a sequence.

You can determine the location of the previous image buffer used by the Image Compression Manager by calling the GetCSequencePrevBuffer function.

You can set a number of compression parameters. Use the SetCSequenceFlushProc function to assign a data-unloading function to the operation. You can set the rate at which the Image Compression Manager inserts key frames into the compressed sequence by calling the SetCSequenceKeyFrameRate function. You can set the frame against which the compressor compares a frame when performing temporal compression by calling the SetCSequencePrev function. Finally, you can control the quality of the compressed image by calling the SetCSequenceQuality function.

SetCSequenceQuality

The SetCSequenceQuality function allows you to adjust the spatial or temporal quality for the current sequence.

```
pascal OSErr SetCSequenceQuality (ImageSequence seqID,
                                  CodecQ spatialQuality,
                                  CodecQ temporalQuality);
```

seqID Contains the unique sequence identifier that was returned by the CompressSequenceBegin function.

`spatialQuality`
Specifies the desired compressed image quality. See“Compression Quality Constants” on page 3-57 for available values.

`temporalQuality`
Specifies the desired sequence temporal quality. This parameter governs the level of compression you desire with respect to information between successive frames in the sequence. Set this parameter to 0 to prevent the compressor from applying temporal compression to the sequence. See “Compression Quality Constants” beginning on page 3-57 for other available values.

DESCRIPTION

The spatial quality parameter indicates the image quality you desire for each frame in the sequence, which governs the level of spatial compression that the compressor may apply to each frame. The temporal quality parameter indicates the sequence quality you desire, which in turn governs the amount of temporal compression that the compressor may apply to the sequence. The new quality parameters take effect with the next frame in the sequence.

You set the default spatial and temporal quality values for a sequence with the `spatialQuality` and `temporalQuality` parameters to the `CompressSequenceBegin` function. For details on `CompressSequenceBegin`, see page 3-106.

If you change the quality settings while processing an image sequence, you affect the maximum image size that you may receive during sequence compression. Consequently, you should call the `GetMaxCompressionSize` function (described on page 3-68) after you change the quality settings. If the maximum size has increased, you should reallocate your image buffers to accommodate the larger image size.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified

SetCSequenceKeyFrameRate

The `SetCSequenceKeyFrameRate` function adjusts the key frame rate for the current sequence.

```
pascal OSErr SetCSequenceKeyFrameRate (ImageSequence seqID,
                                       long keyframerate);
```

`seqID` Contains the unique sequence identifier that was returned by the `CompressSequenceBegin` function (described on page 3-106).

`keyframerate`

Specifies the maximum number of frames allowed between key frames. Key frames provide points from which a temporally compressed sequence may be decompressed. Use this parameter to control the frequency at which the compressor places key frames into the compressed sequence.

The compressor determines the optimum placement for key frames based upon the amount of redundancy between adjacent images in the sequence. Consequently, the compressor may insert key frames more frequently than you have requested. However, the compressor will never place fewer key frames than is indicated by the setting of the `keyFrameRate` parameter. The compressor ignores this parameter if you have not requested temporal compression (that is, you have set the `temporalQuality` parameter to the `CompressSequenceBegin` function to 0).

If you set this parameter to 0, the Image Compression Manager only places key frames in the compressed sequence when you call the `CompressSequenceFrame` function (described on page 3-111) and set the value of the `codecFlagForceKeyFrame` flag to 1 in the `flags` parameter. If you pass in any number other than 0, it specifies the number of non-key frames between key frames. Set this parameter to 1 to specify all key frames, to 2 to specify every other frame as a key frame, to 3 to specify every third frame as a key frame, and so forth.

DESCRIPTION

The key frame rate for a sequence specifies the maximum number of frames allowed between key frames. Key frames provide points from which a temporally compressed sequence may be decompressed. The new key frame rate takes effect with the next image in the sequence. See “Defining Key Frame Rates” on page 3-47 for more information about key frames.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified

SEE ALSO

You set the default key frame rate for a sequence with the `keyFrameRate` parameter to the `CompressSequenceBegin` function (described on page 3-106).

GetCSequenceKeyFrameRate

The `GetCSequenceKeyFrameRate` function lets you determine the current key frame rate of a sequence.

```
pascal OSErr GetCSequenceKeyFrameRate (ImageSequence seqID,
                                       long *keyframerate);
```

`seqID` Contains the unique sequence identifier that was returned by the `CompressSequenceBegin` function (described on page 3-106).

`keyframerate`
Contains a pointer to a long integer that specifies the maximum number of frames allowed between key frames. Key frames provide points from which a temporally compressed sequence may be decompressed.

SEE ALSO

You can set the key frame rate of a sequence with the `SetCSequenceKeyFrameRate` function, described in the previous section.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified

SetCSequenceFrameNumber

The `SetCSequenceFrameNumber` function informs the compressor in use for the specified sequence that frames are being compressed out of order.

```
pascal OSErr SetCSequenceFrameNumber (ImageSequence seqID,
                                      long frameNumber);
```

`seqID` Contains the unique sequence identifier that was returned by the `CompressSequenceBegin` function (described on page 3-106).

`frameNumber`
Specifies the frame number of the frame that is being compressed out of sequence.

DESCRIPTION

This information is only necessary for compressors that are sequence-sensitive.

RESULT CODES

noErr	0	No error
paramErr	–50	Invalid parameter specified

GetCSequenceFrameNumber

The GetCSequenceFrameNumber function returns the current frame number of the specified sequence.

```
pascal OSErr GetCSequenceFrameNumber (ImageSequence seqID,
                                      long *frameNumber);
```

seqID Contains the unique sequence identifier that was returned by the CompressSequenceBegin function (described on page 3-106).

frameNumber
Contains a pointer to the current frame number of the sequence identified by the seqID parameter.

RESULT CODES

noErr	0	No error
paramErr	–50	Invalid parameter specified

SetCSequencePrev

The SetCSequencePrev function allows the application to set the pixel map and boundary rectangle used by the previous frame in temporal compression. This is useful if the application that is compressing has multiple buffers and wants to update the previous frame by switching buffer pointers instead of copying the data. Usually, the Image Compression Manager allocates the previous buffer for temporal compression. Under normal circumstances, the compressor component or the Image Compression Manager updates the contents of the buffer by copying each frame into the buffer after it is compressed.

This is a very specialized function—your application should not need to call it under most circumstances.

```
pascal OSErr SetCSequencePrev (ImageSequence seqID,
                               PixMapHandle prev,
                               const Rect *prevRect);
```

seqID Contains the unique sequence identifier that was returned by the CompressSequenceBegin function (described on page 3-106).

`prev` Contains a handle to the new previous image buffer. The compressor uses this buffer to store a previous image against which the current image (stored in the buffer referred to by the `src` parameter to the `CompressSequenceBegin` function) is compared when performing temporal compression. You must allocate this buffer using the same pixel depth and color table as the source image buffer that you specify with the `src` parameter when you call the `CompressSequenceBegin` function (described on page 3-106). The compressor manages the contents of this buffer based upon several considerations, such as the key frame rate and the degree of difference between compared images.

`prevRect` Contains a pointer to a rectangle defining the portion of the previous image to use for temporal compression. The compressor uses this portion of the previous image as the basis of comparison with the current image. This rectangle must be the same size as the source rectangle you specify with the `srcRect` parameter to the `CompressSequenceBegin` function. To get the boundary of a source pixel map, set this parameter to `nil`.

DESCRIPTION

When you start compressing a sequence, you may assign a previous frame buffer and rectangle with the `prev` and `prevRect` parameters to the `CompressSequenceBegin` function, respectively. If you specified a `nil` value for the `prev` parameter, the compressor allocates an offscreen buffer for the previous frame. In either case you may use this function to assign a new previous frame buffer.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified

Memory Manager errors

SetCSequenceFlushProc

The `SetCSequenceFlushProc` function lets you assign a data-unloading function to a sequence.

```
pascal OSErr SetCSequenceFlushProc (ImageSequence seqID,
                                    FlushProcRecordPtr flushProc,
                                    long bufferSize);
```

`seqID` Contains the unique sequence identifier that was returned by the `CompressSequenceBegin` function (described on page 3-106).

`flushProc` Points to a data-unloading function structure. If there is not enough memory to store the compressed image, the compressor calls a function you provide that unloads some of the compressed data (see "Data-Unloading Functions" beginning on page 3-150 for more

information on the data-unloading structure). If you have not provided a data-unloading function, set this parameter to `nil`. In this case, the compressor writes the entire compressed image into the memory location specified by the `data` parameter to the `CompressSequenceFrame` function (described on page 3-111).

`bufferSize` Specifies the size of the buffer to be used by the data-unloading function specified by the `flushProc` parameter. If you have not specified a data-unloading function, set this parameter to 0.

DESCRIPTION

Data-unloading functions allow compressors to work with images that cannot fit in memory. During the compression operation, the compressor calls the data-unloading function whenever it has accumulated a specified amount of compressed data. Your data-unloading function then writes the compressed data to some other device, freeing buffer space for more compressed data. The compressor starts using the data-unloading function with the next image in the sequence. See "Spooling Compressed Data" on page 3-44 for more information.

There is no parameter to the `CompressSequenceBegin` function (described on page 3-106) that allows you to assign a data-unloading function to a sequence.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified

GetCSequencePrevBuffer

The `GetCSequencePrevBuffer` function determines the location of the previous image buffer allocated by the compressor.

```
pascal OSErr GetCSequencePrevBuffer (ImageSequence seqID,
                                     GWorldPtr *gworld);
```

`seqID` Contains the unique sequence identifier that was returned by the `CompressSequenceBegin` function (described on page 3-106).

`gworld` Contains a pointer to a field to receive a pointer to the structure of type `GWorld` that describes the graphics world for the image buffer. If the compressor has allocated an offscreen image buffer, the compressor returns an appropriate pointer to the graphics world (of type `GWorldPtr`) in the field referred to by this parameter. If the compressor has not allocated a buffer, the function returns an error result code.

You should not dispose of this graphics world—the returned pointer refers to a buffer that the Image Compression Manager is using.

DESCRIPTION

If you do not specify a previous image buffer with the `prev` parameter to the `CompressSequenceBegin` function, the compressor allocates an offscreen graphics world for you. Your program can obtain access to the pixel map in that graphics world by calling this function.

Note that the `GetCSequencePrevBuffer` function only returns information about buffers that were allocated by the compressor. You cannot use this function to determine the location of a buffer you have provided.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified

Constraining Compressed Data

The Image Compression Manager provides two functions and a data structure that allow your application to communicate information to compressors that can constrain compressed data to a specific data rate. Compressors indicate that they can constrain the data rate by setting the following flag in their compressor information structure:

```
#define codecInfoDoesRateConstrain(1L<<23)
```

(For details, see “The Compressor Information Structure” beginning on page 3-52.)

The `DataRateParams` data type defines the data rate parameters structure.

```
typedef struct {
   long     dataRate;              /* bytes per second */
   long     dataOverrun;           /* number of bytes outside
                                      rate */
   long     frameDuration;         /* in milliseconds */
   long     keyFrameRate;          /* frequency of key frames */
   CodecQ   minSpatialQuality;     /* minimum spatial quality */
   CodecQ   minTemporalQuality;    /* minimum temporal quality */
} DataRateParams;
typedef DataRateParams *DataRateParamsPtr;
```

Field descriptions

`dataRate`	Specifies the bytes per second to which the data rate must be constrained.
`dataOverrun`	Indicates the current number of bytes above or below the desired data rate. A value of 0 means that the data rate is being met exactly. If your application doesn’t know the data overrun, it should set this field to 0.
`frameDuration`	Specifies the duration of the current frame in milliseconds.

keyFrameRate Indicates the frequency of key frames. This frequency is normally identical to the key frame rate passed to the CompressSequenceBegin function (described on page 3-106).

minSpatialQuality
Specifies the minimum spatial quality the compressor should use to meet the requested data rate. See "Compression Quality Constants" beginning on page 3-57 for available values.

minTemporalQuality
Indicates the minimum temporal quality the compressor should use to meet the requested data rate. See "Compression Quality Constants" beginning on page 3-57 for available values.

The SetCSequenceDataRateParams function allows you to specify the parameters in this structure and the GetCSequenceDataRateParams function allows you to retrieve the parameters.

SetCSequenceDataRateParams

The SetCSequenceDataRateParams function allows your application to set parameters in the data rate parameters structure, which communicates information to compressors that can constrain compressed data in a particular sequence to a specific data rate.

```
pascal OSErr SetCSequenceDataRateParams
                            (ImageSequence seqID,
                             DataRateParamsPtr params);
```

seqID Contains the unique sequence identifier that was returned by the CompressSequenceBegin function (described on page 3-106).

params Points to the data rate parameters structure to be associated with the sequence identifier specified in the seqID parameter.

DESCRIPTION

If your application is keeping track of data overrun, you should call the SetCSequenceDataRateParams function before each use of the CompressSequenceFrame function (described on page 3-111). If not, you only need to call SetCSequenceDataRateParams before the first use of CompressSequenceFrame, with the dataOverrun parameter of the data rate parameters structure set to 0. In this case, it is assumed that the frame duration is valid for all frames. Setting the dataRate field in the data rate parameters structure to 0 is the same as not performing data rate constraint.

RESULT CODES

noErr	0	No error
paramErr	–50	Invalid parameter specified

GetCSequenceDataRateParams

The GetCSequenceDataRateParams function obtains the data rate parameters previously set with the SetCSequenceDataRateParams function, which is described in the previous section.

```
pascal OSErr GetCSequenceDataRateParams
                            (ImageSequence seqID,
                             DataRateParamsPtr params);
```

seqID Contains the unique sequence identifier that was returned by the CompressSequenceBegin function (described on page 3-106).

params Points to the data rate parameters structure associated with the sequence identifier specified in the seqID parameter.

RESULT CODES

noErr	0	No error
paramErr	–50	Invalid parameter specified

Changing Sequence-Decompression Parameters

This section discusses the functions that enable your application to manipulate the parameters that control sequence decompression and to get information about memory that the decompressor has allocated. Your application establishes the default value for most of these parameters with the DecompressSequenceBegin function (described on page 3-113). Some of these functions deal with parameter values that cannot be set when starting a sequence.

You can determine the buffers used by a decompressor component when it decompresses a sequence. Use the GetDSequenceImageBuffer function to determine the location of the image buffer. Use the GetDSequenceScreenBuffer function to determine the location of the screen buffer.

You can control a number of the parameters that affect a decompression operation (note that changing these parameters may temporarily affect performance). Use the SetDSequenceAccuracy function to control the accuracy of the decompression. Use the SetDSequenceDataProc function to assign a data-loading function to the operation. Use the SetDSequenceMask function to set the clipping region for the resulting image. You can establish a blend matte for the operation by calling the SetDSequenceMatte function. You can alter the spatial characteristics of the resulting image by calling the SetDSequenceMatrix function. Your application can establish the

size and location of the operation's source rectangle by calling the `SetDSequenceSrcRect` function. Finally, you can set the transfer mode used by the decompressor when it draws to the screen by calling the `SetDSequenceTransferMode` function.

SetDSequenceTransferMode

The `SetDSequenceTransferMode` function sets the mode used when drawing the decompressed image.

```
pascal OSErr SetDSequenceTransferMode (ImageSequence seqID,
                                       short mode,
                                       const RGBColor *opColor);
```

`seqID` Contains the unique sequence identifier that was returned by the `DecompressSequenceBegin` function (described on page 3-113).

`mode` Specifies the transfer mode used when drawing the decompressed image. The Image Compression Manager supports the same transfer modes supported by QuickDraw's `CopyBits` routine (described in *Inside Macintosh: Imaging*).

`opColor` Contains a pointer to the color for use in `addPin`, `subPin`, `blend`, and `transparent` operations. The Image Compression Manager passes this color to QuickDraw as appropriate. If `nil`, the opcolor is left unchanged.

DESCRIPTION

The Image Compression Manager supports the same transfer modes supported by QuickDraw's `CopyBits` routine. The new mode takes effect with the next frame in the sequence. For any given sequence, the default opcolor is 50 percent gray and the default mode is `ditherCopy`.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified

SEE ALSO

You set the default transfer mode for a sequence with the `mode` parameter to the `DecompressSequenceBegin` function.

SetDSequenceSrcRect

The `SetDSequenceSrcRect` function defines the portion of the image to decompress.

```
pascal OSErr SetDSequenceSrcRect (ImageSequence seqID,
                                  const Rect *srcRect);
```

`seqID` Contains the unique sequence identifier that was returned by the `DecompressSequenceBegin` function (described on page 3-113).

`srcRect` Contains a pointer to a rectangle defining the portion of the image to decompress. This rectangle must lie within the boundary rectangle of the compressed image, which is defined by (0,0) and `((**desc).width,(**desc).height)`, where `desc` refers to the image description structure you supply to the `DecompressSequenceBegin` function. If the `srcRect` parameter is `nil`, the rectangle is set to the rectangle structure of the image description structure.

DESCRIPTION

The decompressor acts on that portion of the compressed image that lies within this rectangle. The new source rectangle takes effect with the next frame in the sequence.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified

SEE ALSO

You set the default source rectangle for a sequence with the `srcRect` parameter to the `DecompressSequenceBegin` function.

SetDSequenceMatrix

The `SetDSequenceMatrix` function assigns a mapping matrix to the sequence.

```
pascal OSErr SetDSequenceMatrix (ImageSequence seqID,
                                 MatrixRecordPtr matrix);
```

`seqID` Contains the unique sequence identifier that was returned by the `DecompressSequenceBegin` function (described on page 3-113).

`matrix` Points to a matrix structure that specifies how to transform the image during decompression. You can use the matrix structure to translate or scale the image during decompression. To set the matrix to identity, pass `nil` in this parameter. See the chapter "Movie Toolbox" in this book for more information about matrix operations.

DESCRIPTION

The decompressor uses the matrix to create special effects with the decompressed image, such as translating or scaling the image. The new matrix takes effect with the next frame in the sequence.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified

SEE ALSO

You set the default matrix for a sequence with the `matrix` parameter to the `DecompressSequenceBegin` function.

SetDSequenceMask

The `SetDSequenceMask` function assigns a clipping region to the sequence.

```
pascal OSErr SetDSequenceMask (ImageSequence seqID,
                               RgnHandle mask);
```

`seqID` Contains the unique sequence identifier that was returned by the `DecompressSequenceBegin` function (described on page 3-113).

`mask` Contains a handle to a clipping region in the destination coordinate system. If specified, the decompressor applies this mask to the destination image. If you want to stop masking, set this parameter to `nil`.

DESCRIPTION

The decompressor draws only that portion of the decompressed image that lies within the specified clipping region. The new region takes effect with the next frame in the sequence. You should not dispose of this region until the Image Compression Manager is finished with the sequence, or until you set the mask either to `nil` or to a different region by calling the `SetDSequenceMask` function again.

RESULT CODES

noErr	0	No error
paramErr	–50	Invalid parameter specified

Memory Manager errors

SEE ALSO

You set the default clipping region for a sequence with the `mask` parameter to the `DecompressSequenceBegin` function.

SetDSequenceMatte

The `SetDSequenceMatte` function assigns a blend matte to the sequence.

```
pascal OSErr SetDSequenceMatte (ImageSequence seqID,
                                PixMapHandle matte,
                                const Rect *matteRect);
```

`seqID` Contains the unique sequence identifier that was returned by the `DecompressSequenceBegin` function (described on page 3-113).

`matte` Contains a handle to a pixel map that contains a blend matte. You can use the blend matte to cause the decompressed image to be blended into the destination pixel map. The matte can be defined at any supported pixel depth—the matte depth need not correspond to the source or destination depths. However, the matte must be in the coordinate system of the source image. If you want to turn off the blend matte, set this parameter to `nil`.

`matteRect` Contains a pointer to the boundary rectangle for the matte. The decompressor uses only that portion of the matte that lies within the specified rectangle. This rectangle must be the same size as the source rectangle you specify with the `SetDSequenceSrcRect` function (described on page 3-131) or with the `srcRect` parameter to the `DecompressSequenceBegin` function. To specify the matte pixel map bounds, pass `nil` in this parameter.

DESCRIPTION

The decompressor uses the matte to blend the decompressed image into the destination pixel map. The new matte and matte boundary rectangle take effect with the next frame in the sequence. You should not dispose of the matte until the Image Compression Manager is finished with the sequence.

RESULT CODES

noErr	0	No error
paramErr	–50	Invalid parameter specified

Memory Manager errors

SetDSequenceAccuracy

The SetDSequenceAccuracy function adjusts the decompression accuracy for the current sequence.

```
pascal OSErr SetDSequenceAccuracy (ImageSequence seqID,
                                   CodecQ accuracy);
```

seqID Contains the unique sequence identifier that was returned by the DecompressSequenceBegin function (described on page 3-113).

accuracy Specifies the accuracy desired in the decompressed image. Values for this parameter are on the same scale as compression quality. See "Compression Quality Constants" beginning on page 3-57, for available values.

DESCRIPTION

The accuracy parameter governs how precisely the decompressor decompresses the image data. Some decompressors may choose to ignore some image data to improve decompression speed. A new accuracy parameter takes effect with the next frame in the sequence.

RESULT CODES

noErr	0	No error
paramErr	–50	Invalid parameter specified

SEE ALSO

You set the default accuracy value for a sequence with the accuracy parameter to the DecompressSequenceBegin function.

SetDSequenceDataProc

The `SetDSequenceDataProc` function lets you assign a data-loading function to the sequence.

```
pascal OSErr SetDSequenceDataProc (ImageSequence seqID,
                                   DataProcRecordPtr dataProc,
                                   long bufferSize);
```

`seqID` Contains the unique sequence identifier that was returned by the `DecompressSequenceBegin` function (described on page 3-113).

`dataProc` Points to a data-loading function structure. If the data stream is not all in memory when your program calls `DecompressSequenceFrame`, the decompressor calls a function you provide that loads more compressed data (see "Data-Loading Functions" beginning on page 3-149 for more information about data-loading functions). If you have not provided a data-loading function, or if you want the decompressor to stop using your data-loading function, set this parameter to `nil`. In this case, the entire image must be in memory at the location specified by the `data` parameter to the `DecompressSequenceFrame` function (see page 3-116).

`bufferSize` Specifies the size of the buffer to be used by the data-loading function specified by the `dataProc` parameter. If you have not specified a data-loading function, set this parameter to 0.

DESCRIPTION

Data-loading functions allow decompressors to work with images that cannot fit in memory. During the decompression operation the decompressor calls the data-loading function whenever it has exhausted its supply of compressed data. Your data-loading function then fills the available buffer space with more compressed data. The decompressor starts using the data-loading function with the next image in the sequence. See "Spooling Compressed Data," which begins on page 3-44, for more information about data-loading functions.

There is no parameter to the `DecompressSequenceBegin` function that allows you to assign a data-loading function to a sequence.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified

GetDSequenceImageBuffer

The `GetDSequenceImageBuffer` function helps you determine the location of the offscreen image buffer allocated by the decompressor.

```
pascal OSErr GetDSequenceImageBuffer (ImageSequence seqID,
                                      GWorldPtr *gworld);
```

`seqID` Contains the unique sequence identifier that was returned by the `DecompressSequenceBegin` function (described on page 3-113).

`gworld` Contains a pointer to a field to receive a pointer to the structure of type `GWorld` describing the graphics world for the image buffer. If the decompressor has allocated an offscreen image buffer, the decompressor returns an appropriate `GWorldPtr` in the field referred to by this parameter. If the decompressor has not allocated a buffer, the function returns an error result code.

You should not dispose of this graphics world—the returned pointer refers to a buffer that the Image Compression Manager is using. It is disposed of for you when the `CDSequenceEnd` function is called. For details on `CDSequenceEnd`, see page 3-119.

DESCRIPTION

The decompressor uses this buffer when decompressing a sequence that was temporally compressed. You cause the decompressor to use an image buffer by setting the `codecFlagUseImageBuffer` flag to 1 in the `flags` parameter to the `DecompressSequenceBegin` function.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified

GetDSequenceScreenBuffer

The `GetDSequenceScreenBuffer` function enables you to determine the location of the offscreen buffer allocated by the decompressor.

```
pascal OSErr GetDSequenceScreenBuffer (ImageSequence seqID,
                                       GWorldPtr *gworld);
```

`seqID` Contains the unique sequence identifier that was returned by the `DecompressSequenceBegin` function (described on page 3-113).

`gworld` Contains a pointer to a field to receive a pointer to the graphics world structure (defined by the `GWorld` data type) describing the graphics world for the screen buffer. If the decompressor has allocated an offscreen buffer, the decompressor returns an appropriate `GWorldPtr` in the field referred to by this parameter. If the decompressor has not allocated a buffer, the function returns an error result code.

You should not dispose of this graphics world—the returned pointer refers to a buffer that the Image Compression Manager is using. It is disposed of for you when the `CDSequenceEnd` function is called. For details on `CDSequenceEnd`, see page 3-119.

DESCRIPTION

The decompressor uses this buffer for decompressed images. During decompression the decompressor writes the decompressed image to an offscreen buffer and then copies the results to the screen. This reduces the tearing effect that can result from decompressing directly to the screen. You cause the decompressor to use a screen buffer by setting the `codecFlagUseScreenBuffer` flag to 1 in the `flags` parameter to the `DecompressSequenceBegin` function.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified

Working With the StdPix Function

To allow applications to have access to compressed image data as it is displayed, a new graphics function has been added to the `grafProcs` field of the color graphics port structure (defined by the `CGrafPort` data type). See *Inside Macintosh: Imaging* for more information about the color graphics port structure.

The `StdPix` function extends the current `grafProcs` field to support compressed data, mattes, and matrices. The new function supports pixel maps and allows you to intercept image data in compressed form before it is decompressed and displayed. For example, you can use the `StdPix` function to collect compressed image data that is to be processed and printed. In addition, your application can call the `StdPix` function directly.

The replaced `grafProcs` field is referred to in the original QuickDraw documentation as the `newProc1` field. The standard handler is called `StdPix`, and you obtain its address by calling QuickDraw's `SetStdCProcs` routine. Alternatively, your application can call the `StdPix` function directly, using the interface described here. Your application can intercept the low-level `grafProcs` drawing routines just as it would any of the other routines, except that you must call `SetStdCProcs` to gain access to the standard `grafProcs` handler.

Note
QuickDraw's `CopyDeepMask` function uses the `StdPix` function if QuickTime is present. u

See *Inside Macintosh: Imaging* for more information about the QuickDraw low-level drawing routines, the `SetStdCProcs` routine, the `QDProcs` structure, and the `CopyDeepMask` routine.

To work with the control information associated with a compressed image, you can use the `SetCompressedPixMapInfo` and `GetCompressedPixMapInfo` functions (described on page 3-139 and page 3-141, respectively).

StdPix

The Image Compression Manager lets you invoke QuickDraw's `StdPix` function as follows:

```
pascal void StdPix (PixMapPtr src, const  Rect *srcRect,
                    MatrixRecordPtr matrix, short mode,
                    RgnHandle mask, PixMapPtr matte,
                    Rect *matteRect, short flags);
```

`src` Contains a pointer to a pixel map containing the image to draw. Use the `GetCompressedPixMapInfo` function (described on page 3-141) to retrieve information about this pixel map.

`srcRect` Points to a rectangle defining the portion of the image to display. This rectangle must lie within the boundary rectangle of the compressed image or within the source image. If this parameter is set to `nil`, the entire image is displayed.

`matrix` Contains a pointer to a matrix structure that specifies the mapping of the source rectangle to the destination. It is a fixed-point, 3-by-3 matrix. This roughly corresponds to the `dstRect` parameter to QuickDraw's `StdBits` routine. See the chapter "Movie Toolbox" in this book for more information about matrix operations.

`mode` Specifies the transfer mode for the operation. The Image Compression Manager supports the same transfer modes supported by QuickDraw's `CopyBits` routine.

Note that this parameter also controls the accuracy of any decompression operation that may be required to display the image. If bit 7 (0x80) of the `mode` parameter is set to 1, the `StdPix` function sets the decompression accuracy to `codecNormalQuality`. If this bit is set to 0, the function sets the accuracy to `codecHighQuality`.

`mask` Contains a handle to a clipping region in the destination coordinate system. If specified, the compressor applies this mask to the destination image. If there is no mask, this parameter is set to `nil`.

`matte` Points to a pixel map that contains a blend matte. The blend matte causes the decompressed image to be blended into the destination pixel map. The matte can be defined at any supported pixel depth—the matte depth need not correspond to the source or destination depths. However, the matte must be in the coordinate system of the source image. If there is no matte, this parameter is set to `nil`.

The matte may be compressed. Use the `GetCompressedPixMapInfo` function (described on page 3-141) to determine if the matte pixel map contains compressed data.

`matteRect` Contains a pointer to a rectangle defining a portion of the blend matte to apply. This parameter is set to `nil` if there is no matte or if the entire matte is to be used.

`flags` Contains control flags. The following flags are available:

`callOldBits`
If this flag is set, then the `StdPix` function calls QuickDraw's `bitsProc` routine with the decompressed image data. A pointer to this routine is located in the `bitsProc` field of the `CQDProcs` record. If the `bitsProc` routine is not customized, then it is not called unless the `callStdBits` flag is also set. See the description of the `CQDProcs` record in *Inside Macintosh: Imaging* for more on the `bitsProc` routine.

`callStdBits`
If this flag is set, the `callOldBits` flag is set, and the `CQDProcs` record's `bitsProc` field is set to the `StdBits` routine, then the `StdBits` routine is called with the decompressed image data.

`noDefaultOpcodes`
If this flag is set and a picture is open for writing, the default picture opcodes (for displaying a warning when QuickTime is not installed) are not added to the output picture. This can be useful when storing multiple `StdPix` opcodes in a single picture.

SetCompressedPixMapInfo

The `SetCompressedPixMapInfo` function allows your application to store information about a compressed image for the `StdPix` function (described in the previous section).

```
pascal OSErr SetCompressedPixMapInfo (PixMapPtr pix,
                            ImageDescriptionHandle desc,
                            Ptr data, long bufferSize,
                            DataProcRecordPtr dataProc,
                            ProgressProcRecordPtr progressProc);
```

`pix` Points to a structure that holds encoded compressed image data.

`desc` Contains a handle to the image description structure that defines the compressed image.

`data` Points to the buffer for the compressed image data. If the entire compressed image cannot be stored at this location, you may assign a data-loading function (see the discussion of the `dataProc` parameter to this function). This pointer must contain a 32-bit clean address.

`bufferSize`
Specifies the size of the buffer to be used by the data-loading function specified by the `dataProc` parameter. If there is no data-loading function defined for this operation, set this parameter to 0.

`dataProc` Points to a data-loading function structure. If there is not enough memory to store the compressed image, the decompressor calls a function you provide that loads more compressed data (see "Data-Loading Functions" beginning on page 3-149 for more information about data-loading functions). If you do not want to assign a data-loading function, set this parameter to `nil`.

`progressProc`
Points to a progress function structure. During the decompression operation, the decompressor may occasionally call a function you provide in order to report its progress (see "Progress Functions" beginning on page 3-152 for more information about progress functions). If you do not want to assign a progress function, set this parameter to `nil`. If you pass a value of –1, you obtain a standard progress function.

DESCRIPTION

The `SetCompressedPixMapInfo` function stores information in a structure that is identical to a `PixMap` structure, but the structure represents the compressed data, not the actual pixel map. You can use the `SetCompressedPixMapInfo` if you are working with the `StdPix` function (described on page 3-138).

RESULT CODES

`paramErr`	–50	Invalid parameter specified

SEE ALSO

You can retrieve information about a compressed pixel map by calling the `GetCompressedPixMapInfo` function, which is described in the next section.

GetCompressedPixMapInfo

The `GetCompressedPixMapInfo` function allows your application to retrieve information about a compressed image.

```
pascal OSErr GetCompressedPixMapInfo (PixMapPtr pix,
                            ImageDescriptionHandle *desc,
                            Ptr *data, long *bufferSize,
                            DataProcRecord *dataProc,
                            ProgressProcRecord *progressProc);
```

`pix` Points to a structure that holds encoded compressed image data.

`desc` Contains a pointer to a field that is to receive a handle to the image description structure that defines the compressed image. If you are not interested in this information, you may specify `nil` in this parameter.

`data` Contains a pointer to a field that is to receive a pointer to the compressed image data. If the entire compressed image cannot be stored at this location, you can define a data-loading function for this operation (see the discussion of the `dataProc` parameter to this function). If you are not interested in this information, you may specify `nil` in this parameter.

`bufferSize`
Contains a pointer to a field that is to receive the size of the buffer to be used by the data-loading function specified by the `dataProc` parameter. If there is no data-loading function defined for this operation, this parameter is ignored. If you are not interested in this information, you may specify `nil` in this parameter.

`dataProc` Contains a pointer to a data-loading function structure. If there is not enough memory to store the compressed image, the decompressor calls a function you provide that loads more compressed data (see "Data-Loading Functions" beginning on page 3-149 for more information about data-loading functions). If there is no data-loading function for this image, the function sets the `dataProc` field in the function structure to `nil`. If you are not interested in this information, you may specify `nil` in this parameter.

`progressProc`
Contains a pointer to a progress function structure. During a decompression operation, the decompressor may occasionally call a function you provide in order to report its progress (see "Progress Functions" beginning on page 3-152 for more information about progress functions). If there is no progress function for this image, the function sets the `progressProc` field in the function structure to `nil`. If you pass a value of –1, you obtain a standard progress function. If you are not interested in this information, you may specify `nil` in this parameter.

DESCRIPTION

The data in the compressed image has been encoded in a `PixMap` structure with the `SetCompressPixMapInfo` function. This is the kind of pixel map that may be passed into the `StdPix` function. If you pass a normal, non-encoded pixel map, `GetCompressedPixMapInfo` returns a `paramErr` result code. You use the `GetCompressedPixMapInfo` function if you are intercepting calls to the `StdPix` function.

SPECIAL CONSIDERATIONS

The pixel map structure filled in by the `GetCompressedPixMapInfo` function should not be used by any other Macintosh functions. It is only to be used by the `StdPix` function.

RESULT CODES

`paramErr`	–50	Invalid parameter specified

SEE ALSO

You can set information about a compressed pixel map by calling the `SetCompressedPixMapInfo` function, which is described in the previous section.

Aligning Windows

This section describes the functions that allow your application to position and drag windows to optimal screen positions based on the depth of the screen. These functions are useful for movie playback performance considerations that depend on where you draw on the screen.

The Image Compression Manager places the windows at an optimal position on the screen by aligning rectangles horizontally on 1-bit and 2-bit screens to multiples of 16 pixels, aligning 4-bit screens to multiples of 8, aligning 8-bit screens to multiples of 4, and aligning 16-bit screens to multiples of 2. (Alignment on 32-bit screens is to multiples of 4 pixels and only occurs on Macintosh computers of class 68040 or greater.) When the alignment rectangle crosses more than one screen, the Image Compression Manager uses the alignment of the strictest screen.

Decompression to non-optimally aligned destinations can reduce performance by as much as 50 percent, so you should use these functions whenever possible.

The alignment behavior provided by these functions is adequate in the vast majority of situations. However, if you need customized alignment behavior (for example, justification specifications geared to particular video hardware), you can use the application-defined function described in “Alignment Functions” on page 3-155 to override the standard alignment. See the chapter “Sequence Grabber Components” in *Inside Macintosh: QuickTime Components* for more information on application-defined

alignment functions and video hardware. All the alignment functions provide a parameter in which you can specify a function with customized alignment behavior.

The `AlignWindow` function enables you to transport a specified window to the nearest optimal alignment position. The `DragAlignedWindow` function drags the specified window along an optimal alignment grid. The `DragAlignedGrayRgn` function drags a specified gray region along an optimal alignment grid. The `AlignScreenRect` function aligns a specified rectangle to the strictest screen that the rectangle intersects.

AlignWindow

The `AlignWindow` function moves a specified window to the nearest optimal alignment position.

```
pascal void AlignWindow (WindowPtr wp, Boolean front,
                         const Rect *alignmentRect,
                         AlignmentProcRecordPtr alignmentProc);
```

`wp` Points to the window to be aligned.

`front` Specifies the frontmost window. If the front parameter is `true` and the window specified in the `wp` parameter isn't the active window, `AlignWindow` makes it the active window by calling the Window Manager's `SelectWindow` routine.

`alignmentRect`
Contains a pointer to a rectangle in window coordinates that allows you to align the window to a rectangle within the window. Set this parameter to `nil` to align using the bounds of the window.

`alignmentProc`
Points to a function that allows you to provide your own alignment behavior. Set this parameter to `nil` to use the standard behavior. Your alignment function must be in the following form:

```
pascal void MyAlignmentProc(Rect *rp, long refcon);
```

See "Alignment Functions" on page 3-155 for details.

SEE ALSO

The `AlignWindow` function is similar to the Window Manager's `MoveWindow` routine. See *Inside Macintosh: Macintosh Toolbox Essentials* for details.

DragAlignedWindow

The `DragAlignedWindow` function drags the specified window along an optimal alignment grid.

```
pascal void DragAlignedWindow (WindowPtr wp, Point startPt,
                               Rect *boundsRect,
                               Rect *alignmentRect,
                               AlignmentProcRecordPtr alignmentProc);
```

`wp` Contains a window pointer to the window to be dragged.

`startPt` Specifies a point that is equal to the point where the mouse button was pressed (in global coordinates, as stored in the `where` field of the event structure). `DragAlignedWindow` pulls a gray outline of the window around the screen, following the movements of the mouse until the button is released.

`boundsRect`
Points to the boundary rectangle in global coordinates. If the mouse button is released when the mouse position is outside the limits of the boundary rectangle, `DragAlignedWindow` returns without moving the window or making it the active window. For a document window, the boundary rectangle typically is four pixels in from the menu bar and from the other edges of the screen, to ensure that there won't be less than a four-pixel-square area of the title bar visible on the screen.

`alignmentRect`
Points to a rectangle in window coordinates that allows you to align the window to a rectangle within the window. Set this parameter to `nil` to align using the bounds of the window.

`alignmentProc`
Allows you to provide your own alignment behavior. Set this parameter to `nil` to use the standard alignment behavior. Your alignment function must be in the following form:

```
pascal void MyAlignmentProc (Rect *rp, long refcon);
```

See "Alignment Functions" on page 3-155 for details.

SEE ALSO

The `DragAlignedWindow` is similar to the Window Manager's `DragWindow` routine. See *Inside Macintosh: Macintosh Toolbox Essentials* for details on `DragWindow`.

DragAlignedGrayRgn

The `DragAlignedGrayRgn` function drags the specified gray region along an optimal alignment grid.

```
pascal long DragAlignedGrayRgn (RgnHandle theRgn, Point startPt,
                                Rect *boundsRect, Rect *slopRect,
                                short axis, ProcPtr actionProc,
                                Rect *alignmentRect,
                                AlignmentProcRecordPtr alignmentProc);
```

`theRgn` Contains a region handle to the specified region for this operation. When the user holds down the mouse button, `DragAlignedGrayRgn` pulls a gray outline of the region around following the movement of the mouse until the mouse button is released.

`startPt` Specifies the point where the mouse button was originally pressed in the local coordinates of the current graphics port.

`boundsRect`
Contains a pointer to the boundary rectangle of the current graphics port. The offset point follows the mouse location except that `DragAlignedGrayRgn` never moves the offset point outside this rectangle. This limits the travel of the region's outline, not the movements of the mouse.

`slopRect` Contains a pointer to the `slop` rectangle that completely encloses the boundary rectangle so that the user is allowed some flexibility in moving the mouse.

`axis` Allows you to constrain the region's motion to only one axis. Set this parameter to 0 to specify no constraint. To indicate constraint along a horizontal axis, set this parameter to 1. To indicate constraint along a vertical axis, set this parameter to 2. See *Inside Macintosh: Macintosh Toolbox Essentials* for details on the constants for the axis parameter of the Window Manager's `DragGrayRgn` routine.

`actionProc`
Points to a function that defines some action to be performed repeatedly as long as the user holds down the mouse button. The function should have no parameters. If the `actionProc` parameter is `nil`, `DragAlignedGrayRgn` simply retains control until the mouse button is released.

`alignmentRect`
Contains a pointer to a rectangle within the bounds of the region specified in the parameter `theRgn`. Pass `nil` to align using the bounds of the parameter `theRgn`.

`alignmentProc`
Points to your own alignment behavior function. Pass `nil` to use the standard behavior. Your alignment function must be in the following form:

```
pascal void MyAlignmentProc (Rect *rp, long refcon);
```

See "Alignment Functions" on page 3-155 for details.

DESCRIPTION

The `DragAlignedGrayRgn` function is not normally made directly. The `DragAlignedWindow` function (described on page 3-144) calls this function.

SEE ALSO

The `DragAlignedGrayRgn` function is nearly identical to the Window Manager's `DragGrayRgn` routine. See *Inside Macintosh: Macintosh Toolbox Essentials* for details on `DragGrayRgn`.

AlignScreenRect

The `AlignScreenRect` function aligns a specified rectangle to the strictest screen that the rectangle intersects.

```
pascal void AlignScreenRect (Rect *rp,
                             AlignmentProcRecordPtr alignmentProc);
```

`rp` Contains a pointer to a rectangle defined in global screen coordinates.

`alignmentProc`
Points to your own alignment behavior function. Set this parameter to `nil` to use the standard behavior. Your alignment function must be in the following form:

```
pascal void MyAlignmentProc (Rect *rp, long refCon);
```

See "Alignment Functions" on page 3-155 for details.

DESCRIPTION

Normally, the `AlignScreenRect` function is not called directly.

Working With Graphics Devices and Graphics Worlds

This section describes two Image Compression Manager functions that enable you to select graphics devices and create graphics worlds. You can use the `GetBestDeviceRect` function to select the best available graphics device. The `NewImageGWorld` function allows you to create a graphics world based on the width, height, depth, and color table of a specified image description structure.

GetBestDeviceRect

The `GetBestDeviceRect` function selects the deepest of all available graphics devices, while treating 16-bit and 32-bit screens as having equal depth.

```
pascal OSErr GetBestDeviceRect (GDHandle *gdh, Rect *rp);
```

`gdh` Contains a pointer to the handle of the rectangle for the chosen device. If you do not need the information in this parameter returned, specify `nil`.

`rp` Contains a pointer to the rectangle that is adjusted for the height of the menu bar if the device is the main device. If you do not need the information in this parameter returned, specify `nil`.

DESCRIPTION

If multiple 16-bit and 32-bit monitors are available, the `GetBestDeviceRect` function selects the 16-bit or 32-bit device upon which the cursor has currently been detected. If a cursor is not on one of the devices in question, the first of those in the list is chosen.

Note that the `GetBestDeviceRect` function does not center a rectangle on a device. Rather, it returns the rectangle for the best device.

NewImageGWorld

The `NewImageGWorld` function creates a graphics world from the width, height, depth, and color table of a specified image-description structure.

```
pascal QDErr NewImageGWorld (GWorldPtr *gworld,
                             ImageDescription **idh,
                             GWorldFlags flags);
```

`gworld` Contains a pointer to a graphic world created using the width, height, depth, and color table specified in the image description structure pointed to in the `idh` parameter.

`idh`	Contains a handle to an image description structure with information for the graphics world pointed to by the `gworld` parameter.
`flags`	Contains graphics world flags. These flags are passed directly through to the `NewGWorld` function. (For details on `NewGWorld`, see *Inside Macintosh: Devices*.)

DESCRIPTION

The `NewImageGWorld` function selects the appropriate color table using the `depth` field or custom color table in the image description structure. It creates a 32-bit-deep graphics world if the depth specified in the image description structure is 24.

SPECIAL CONSIDERATIONS

You are responsible for disposing of the graphics world with the `DisposeGWorld` routine. (For more on `DisposeGWorld`, see *Inside Macintosh: Devices*.)

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`cDepthErr`	–157	Invalid pixel resolution

Application-Defined Functions

This section describes four callback functions that you may provide to compressor components and an application-defined function that specifies alignment behavior.

The Image Compression Manager defines four callback functions that applications may provide to compressors or decompressors. These callbacks are data-loading functions, data-unloading functions, completion functions, and progress functions.

- Data-loading functions and data-unloading functions support spooling of compressed data.
- Completion functions allow compressors and decompressors to report that asynchronous operations have completed.
- Progress functions provide a mechanism for compressors and decompressors to report their progress toward completing an operation.

This section describes the interfaces presented when compressors invoke your callback functions. These application-defined functions may be called by compressor components during a compression or decompression operation.

You identify a callback function to an Image Compression Manager function by specifying a pointer to a callback function structure. These structures contain two fields: a pointer to the callback function and a reference constant value. There is one callback function structure for each type of callback function. See the individual function descriptions in the sections that follow for descriptions of the structures.

Data-Loading Functions

Compressors use the data-loading and data-unloading functions when working with images that do not fit into memory. The data-loading function supplies compressed data during a decompression operation.

The `DataProcPtr` data type defines a pointer to a data-loading function. You assign a data-loading function to an image or a sequence by passing a pointer to a structure that identifies the function to the appropriate decompress function.

```
/* data-loading function structure */
typedef struct DataProcRecord DataProcRecord;
typedef DataProcRecord *DataProcRecordPtr;
```

The data-loading function structure contains the following fields:

```
struct DataProcRecord
{
   DataProcPtr dataProc;   /* pointer to data-loading function */
   long        dataRefCon; /* reference constant */
};
```

Field descriptions

`dataProc` Contains a pointer to your data-loading function.

`dataRefCon` Contains a reference constant for use by your data-loading function.

DESCRIPTION

If your data-loading function returns a nonzero result code, the Image Compression Manager terminates the current operation.

MyDataLoadingProc

Your data-loading function should have the following form:

```
pascal OSErr MyDataLoadingProc (Ptr *dataP, long bytesNeeded,
                                long refcon);
```

`dataP` Contains a pointer to the address of the data buffer. The decompressor uses this parameter to indicate where your data-loading function should return the compressed data. You establish this data buffer when you start the decompression operation. For example, the `data` parameter to the `FDecompressImage` function (described on page 3-79) defines the location of the data buffer for that operation. Upon return from your data-loading function, this pointer should refer to the beginning of the compressed data that you loaded.

The decompressor may also use this parameter to indicate that it wants to reset the mark within the compressed data stream. If the `dataP` parameter is set to `nil`, the `bytesNeeded` parameter contains the new mark position, relative to the current position of the data stream. If your data-loading function does not support this operation, return a nonzero result code.

`bytesNeeded`

Specifies the number of bytes requested or the new mark offset. If the decompressor has requested additional compressed data (that is, the value of the `dataP` parameter is not `nil`), then this parameter specifies how many bytes to return. This value never exceeds the size of the original data buffer. Your data-loading function should read the data from the current mark in the input data stream.

If the decompressor has requested to set a new mark position in the data stream (that is, the value of the `dataP` parameter is `nil`), then this parameter specifies the new mark position relative to the current position of the data stream.

`refcon` Contains a reference constant value for use by your data-loading function. Your application specifies the value of this reference constant in the data-loading function structure you pass to the Image Compression Manager.

SPECIAL CONSIDERATIONS

The pointer in the `dataP` parameter must contain a 32-bit clean address within the data buffer. If you have dereferenced a handle, you should call the Memory Manager's `StripAddress` routine before passing it to the `MyDataLoadingProc` function.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`codecSpoolErr`	–8966	Error loading or unloading data

Data-Unloading Functions

Compressors use the data-loading and data-unloading functions when working with images that do not fit into the computer's memory. The data-unloading function writes compressed data to a storage device during a compression operation.

The `FlushProcPtr` data type defines a pointer to a data-unloading function.

```
/* data-unloading structure */
typedef struct FlushProcRecord FlushProcRecord;
typedef FlushProcRecord *FlushProcRecordPtr;
```

You assign a data-unloading function to an image or a sequence by passing a pointer to a structure that identifies the function to the appropriate compression function.

The data-unloading function structure contains the following fields:

```
struct FlushProcRecord
{
   FlushProcPtr flushProc;/* pointer to data-unloading function */
   long        flushRefCon;/* reference constant */
};
```

Field descriptions

`flushProc` Contains a pointer to your data-unloading function.

`flushRefCon` Contains a reference constant for use by your data-unloading function.

MyDataUnloadingProc

Your data-unloading function should have the following form:

```
pascal OSErr MyDataUnloadingProc (Ptr data, long bytesAdded,
                                  long refcon);
```

`data` Points to the data buffer. The compressor uses this parameter to indicate where your data-unloading function can find the compressed data. You establish this data buffer when you start the compression operation. For example, the `data` parameter to the `FCompressImage` function (described on page 3-75) defines the location of the data buffer for that operation. This pointer contains a 32-bit clean address. Your data-unloading function should make no other assumptions about the value of this address.

The compressor may also use this parameter to indicate that it wants to reset the mark within the compressed data stream. If the `data` parameter is set to `nil`, the `bytesNeeded` parameter contains the new mark position, relative to the current position of the output data stream. If your data-unloading function does not support this operation, return a nonzero result code.

`bytesAdded`

Specifies the number of bytes to write or the new mark offset. If the compressor wants to write out some compressed data (that is, the value of `data` is not `nil`), then this parameter specifies how many bytes to write. This value never exceeds the size of the original data buffer. Your data-unloading function should write that data at the current mark in the output data stream.

If the compressor has requested to set a new mark position in the output data stream (that is, the value of `data` is `nil`), then this parameter specifies the new mark position relative to the current position of the data stream.

`refcon` Contains a reference constant value for use by your data-unloading function. Your application specifies the value of this reference constant in the data-unloading function structure you pass to the Image Compression Manager.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`codecSpoolErr`	–8966	Error loading or unloading data

Progress Functions

Compressors and decompressors call progress functions to report on their progress in the current operation. When a component calls your progress function, it supplies you with a number that indicates the completion percentage. This fixed-point value may range from 0.0 through 1.0. Your program can cause the component to terminate the current operation by returning a result code of `codecAbortErr`.

The Image Compression Manager calls your progress function only during long operations, and it does not call your function more than 30 times per second.

The `ProgressProcPtr` data type defines a pointer to a progress function. You assign a progress function to an image or a sequence by passing a pointer to a structure that identifies the progress function to the appropriate function.

```
/* progress function structure */
typedef struct ProgressProcRecord ProgressProcRecord;
typedef ProgressProcRecord *ProgressProcRecordPtr;
```

The progress function structure contains the following fields:

```
struct ProgressProcRecord
{
   ProgressProcPtr progressProc; /* ptr to progress function */
   long           progressRefCon;/* reference constant */
};
```

Field descriptions

`progressProc` Contains a pointer to your progress function.

`progressRefCon`
Contains a reference constant for use by your progress function.

MyProgressProc

Your progress function should have the following form:

```
pascal OSErr MyProgressProc (short message, Fixed completeness,
                             long refcon);
```

`message` Indicates why the Image Compression Manager called your function. The following values are valid:

`codecProgressOpen`
Indicates the start of a long operation. This is always the first message sent to your function. Your function can use this message to trigger the display of your progress window.

`codecProgressUpdatePercent`
Passes completion information to your function. The Image Compression Manager repeatedly sends this message to your function. The `completeness` parameter indicates the relative completion of the operation. You can use this value to update your progress window.

`codecProgressClose`
Indicates the end of a long operation. This is always the last message sent to your function. Your function can use this message as an indication to remove its progress window.

`completeness`
Contains a fixed-point value indicating how far the operation has progressed. Its value is always between 0.0 and 1.0. This parameter is valid only when the message field is set to `codecProgressUpdatePercent`.

`refcon` Contains a reference constant value for use by your progress function. Your application specifies the value of this reference constant in the progress function structure you pass to the Image Compression Manager.

DESCRIPTION

The following functions have parameters that allow you to provide application-defined progress functions: `FCompressImage`, `FDecompressImage`, `TrimImage`, `FCompressPicture`, `FCompressPictureFile`, `DrawPictureFile`, `DrawTrimmedPicture`, `DrawTrimmedPictureFile`, `MakeThumbNailFromPicture`, `MakeThumbnailFromPictureFile`, `MakeThumbnailFromPixMap`, `SetCompressedPixMapInfo`, and `GetCompressedPixMapInfo`. If you pass a value of –1 in the `progressProc` parameter of any of these functions, you obtain a standard progress function.

RESULT CODES

noErr	0	No error
paramErr	–50	Invalid parameter specified
codecAbortErr	–8967	Operation aborted by the progress function

Completion Functions

Compressor components call completion functions when they have finished an asynchronous operation. The component supplies a result code to your completion function. This result code indicates the success or failure of the asynchronous operation. Note that any other result data that may be produced by the asynchronous operation is not valid until the component calls your completion function.

The `CompletionProcPtr` data type defines a pointer to a completion function. You assign a completion function to an image or a sequence by passing a pointer to a structure that identifies the function to the appropriate function.

```
typedef struct CompletionProcRecord CompletionProcRecord;
```

The completion function structure contains the following fields:

```
typedef CompletionProcRecord *CompletionProcRecordPtr;

struct CompletionProcRecord
{
   CompletionProcPtr completionProc;
                              /* pointer to completion function */
   long              completionRefCon;
                              /* reference constant */
};
```

Field descriptions

`completionProc`
Contains a pointer to your completion function. Your completion function may be called at interrupt time. Therefore, the value of the A5 register is unknown, and your function may not use Memory Manager functions or other functions that move memory.

`completionRefCon`
Contains a reference constant for use by your completion function.

MyCompletionProc

Your completion function should have the following form:

```
pascal OSErr MyCompletionProc (OSErr result, short flag,
                               long refcon);
```

`result` Indicator of success of current operation.

`flag` Indicates which part of the operation is complete. The following flags are defined:

`codecCompletionSource`
The Image Compression Manager is done with the source buffer. The Image Compression Manager sets this flag to 1 when it is done with the processing associated with the source buffer. For compression operations, the source is the uncompressed pixel map you are compressing. For decompression operations, the source is the decompressed data you are decompressing.

`codecCompletionDest`
The Image Compression Manager is done with the destination buffer. The Image Compression Manager sets this flag to 1 when it is done with the processing associated with the destination buffer.

Note that more than one of these flags may be set to 1.

`refcon` Contains a reference constant value for use by your completion function. Your application specifies the value of this reference constant in the callback function structure you pass to the Image Compression Manager.

RESULT CODES

`noErr`	0	No error

Alignment Functions

Your application can use alignment functions to specify the alignment in any of the Image Compression Manager's alignment functions (described in "Aligning Windows" beginning on page 3-142). You call the alignment function with a rectangle (defined in global screen coordinates) that has already been aligned using the default behavior. The alignment function then has the option of applying some additional alignment criteria to the rectangle, such as vertical alignment of some form. In the case of supporting hardware alignment, it is the function's responsibility to determine if the rectangle applies to the relevant device.

The `AlignmentProcPtr` data type defines a pointer to an alignment function. You assign an alignment function by passing a pointer to the alignment function structure, which identifies the alignment function to the appropriate function.

```
/* alignment function structure */
typedef struct
{
   AlignmentProcPtr  alignmentProc;    /* pointer to your
                                          alignment function */
   long              alignmentRefCon;  /* reference constant */
} AlignmentProcRecord, *AlignmentProcRecordPtr;
```

Field descriptions

`alignmentProc` Points to your alignment function.

`alignmentRefCon`
Contains a reference constant for use by your alignment function.

MyAlignmentProc

Your alignment function should have the following form:

```
pascal void MyAlignmentProc (Rect *rp, long refcon);
```

`rp` Contains a pointer to a rectangle that has already been aligned with a default alignment function.

`refcon` Contains a reference constant value for use by your alignment function. Your application specifies the value of this reference constant in the alignment function structure you pass to the Image Compression Manager.

Summary of the Image Compression Manager

C Summary

Constants

```
/* determines if Image Compression Manager is available */
#define gestaltCompressionMgr 'icmp'

/* smallest data buffer you may allocate for image data spooling */
#define codecMinimumDataSize 32768

/* compressor component type */
#define compressorComponentType 'imco'

/* decompressor component type */
#define decompressorComponentType 'imdc'

/* Image Compression Manager function control flags */
#define codecFlagUseImageBuffer (1L<<0)   /* (input) use image buffer */
#define codecFlagUseScreenBuffer(1L<<1)   /* (input) use screen buffer */
#define codecFlagUpdatePrevious (1L<<2)   /* (input) update previous
                                             buffer */
#define codecFlagNoScreenUpdate (1L<<3)   /* (input) don't update screen */
#define codecFlagWasCompressed  (1L<<4)   /* (input) image compressed */
#define codecFlagDontOffscreen  (1L<<5)   /* don't go offscreen
                                             automatically */
#define codecFlagUpdatePreviousComp (1L<<6)
                                          /* (input) update previous
                                             buffer */
#define codecFlagForceKeyFrame (1L<<7)    /* force key frame from image */
#define codecFlagOnlyScreenUpdate
                                (1L<<8)   /* decompress current frame */
#define codecFlagLiveGrab  (1L<<9)        /* sequence from live video grab */
#define codecFlagDontUseNewImageBuffer (1L<<10)
                                          /* (input) don't use new image
                                             buffer */
```

```
#define codecFlagInterlaceUpdate (1L<<11)
                                        /* (input) update screen
                                            interlacing */

/*
   status flags from outflags parameter of DecompressSequenceFrame
   function
*/
#define  codecFlagUsedNewImageBuffer (1L<<14)
                                       /* (output) used new image buffer */
#define  codecFlagUsedImageBuffer (1L<<15)
                                       /* (output) used image buffer */

/* completion flags from application-defined completion functions */
#define  codecCompletionSource (1<<0) /* Image Compression Manager done
                                        with source buffer */
#define  codecCompletionDest   (1<<1)/* Image Compression Manager done with
                                        destination buffer */

/* compression quality values */
#define codecMinQuality       0x000L /* minimum-quality image reproduction */
#define codecLowQuality       0x100L /* low-quality image reproduction */
#define codecNormalQuality    0x200L /* normal-quality image reproduction */
#define codecHighQuality      0x300L /* high-quality image reproduction */
#define codecMaxQuality       0x3FFL /* maximum-quality image reproduction */
#define codecLosslessQuality  0x400L /* lossless-quality reproduction */

/*
   special compressor and decompressor identifiers let you choose an
   image compressor component
*/
#define anyCodec              (CodecComponent)0)   /* first one or a
                                                      specified type */
#define bestSpeedCodec        ((CodecComponent)-1) /* fastest of specified
                                                      type */
#define bestFidelityCodec     (CodecComponent)-2)  /* most accurate of
                                                      specified type */
#define bestCompressionCodec( (CodecComponent)-3)  /* one with smallest
                                                      resulting data */

/*
   constants for doDither parameter of DrawTrimmedPictureFile and
   FCompressPictureFile functions
*/
```

```
#define defaultDither0     /* respect dithering instructions in
                               source picture */
#define forceDither1       /* dither image */
#define suppressDither2    /* don't dither image */
```

Data Types

```
typedef Component CompressorComponent; /* compressor identifier */
typedef Component DecompressorComponent;/* decompressor identifier */
typedef Component CodecComponent;      /* compressor identifier */

typedef long CodecType;                /* compressor type */

typedef unsigned short CodecFlags;     /* compressor component flags */

typedef unsigned long CodecQ;          /* compression quality */

typedef pascal OSErr (*DataProcPtr) (Ptr *dataP, long bytesNeeded,
                      longrefCon);  /* pointer to a data-loading function */

typedef pascal OSErr (*FlushProcPtr) (Ptr data, long bytesAdded,
                      long refCon);/* pointer to a data-unloading function */

typedef pascal void (*CompletionProcPtr)(OSErr result, short flags,
                      long refCon);/* pointer to a completion function */

typedef pascal OSErr (*ProgressProcPtr)(short message, Fixed completeness,
                      long refCon);/* pointer to a progress function */

typedef long ImageSequence;      /* unique sequence identifier */

/* progress function structure */
struct ProgressProcRecord
{
      ProgressProcPtr progressProc; /* pointer to your progress function */
      long progressRefCon;          /* reference constant */
};
typedef struct ProgressProcRecord ProgressProcRecord;
typedef ProgressProcRecord *ProgressProcRecordPtr;

/* completion function structure */
struct CompletionProcRecord
{
```

```
      CompletionProcPtr completionProc;/* pointer to completion function */
      long completionRefCon;           /* reference constant */
};
typedef struct CompletionProcRecord CompletionProcRecord;
typedef CompletionProcRecord *CompletionProcRecordPtr;

/* data-loading structure */
struct DataProcRecord
{
      DataProcPtr dataProc;            /* pointer to data-loading function */
      long dataRefCon;                 /* reference constant */
};
typedef struct DataProcRecord DataProcRecord;
typedef DataProcRecord *DataProcRecordPtr;

/* data-unloading structure */
struct FlushProcRecord
{
      FlushProcPtr flushProc; /* pointer to data-unloading function */
      long flushRefCon;       /* reference constant */
};
typedef struct FlushProcRecord FlushProcRecord;
typedef FlushProcRecord *FlushProcRecordPtr;

typedef pascal void (*StdPixProcPtr)(PixMap *src, Rect *srcRect,
                                     MatrixRecord *matrix,
                                     short mode, RgnHandle mask,
                                     PixMap *matte,Rect *matteRect,
                                     short flags);

typedef struct
{
   AlignmentProcPtr     alignmentProc;    /* pointer to your alignment
                                             function */
   long                 alignmentRefCon;  /* reference constant */
} AlignmentProcRecord;

typedef AlignmentProcRecord *AlignmentProcRecordPtr;

typedef struct
{
   long     dataRate;              /* bytes per second */
   long     dataOverrun;           /* number of bytes outside rate */
   long     frameDuration;         /* in milliseconds */
```

```
   long     keyFrameRate;           /* frequency of key frames */
   CodecQ   minSpatialQuality;      /* minimum spatial quality */
   CodecQ   minTemporalQuality;     /* minimum temporal quality */
} DataRateParams;
typedef DataRateParams *DataRateParamsPtr;

/* image description structure */
struct ImageDescription
{
      long idSize;            /* total size of this structure */
      CodecType cType;        /* compressor type of creator */
      long resvd1;            /* reserved--must be set to 0 */
      short resvd2;           /* reserved--must be set to 0 */
      short dataRefIndex;     /* reserved--must be set to 0 */
      short version;          /* version of compressed data */
      short revisionLevel;    /* version of compressor that created data */
      long vendor;            /* developer of compressor that created data */
      CodecQ temporalQuality; /* degree of temporal compression */
      CodecQ spatialQuality;  /* degree of spatial compression */
      short width;            /* width of source image in pixels */
      short height;           /* height of source image in pixels */
      Fixed hRes;             /* horizontal resolution of source image */
      Fixed vRes;             /* vertical resolution of source image */
      long dataSize;          /* size in bytes of compressed data */
      short frameCount;       /* number of frames in image data */
      Str31 name;             /* name of compression algorithm */
      short depth;            /* pixel depth of source image */
      short clutID;           /* ID number of color table for image */
};
typedef struct ImageDescription ImageDescription;
typedef ImageDescription *ImageDescriptionPtr, **ImageDescriptionHandle;

/* compressor information structure */
struct CodecInfo
{
      Str31 typeName;           /* compression algorithm */
      short version;            /* version of compressed data */
      short revisionLevel;      /* version of component */
      long vendor;              /* developer of component */
      long decompressFlags;     /* decompression capability flags */
      long compressFlags;       /* compression capability flags */
      long formatFlags;         /* compression format flags */
      unsigned char compressionAccuracy;
                                /* relative accuracy of compression */
```

```
      unsigned char decompressionAccuracy;
                                /* relative accuracy of decompression */
      unsigned short compressionSpeed;
                                /* relative speed of compressor */
      unsigned short decompressionSpeed;
                                /* relative speed of decompressor */
      unsigned char compressionLevel;
                                /* relative level of compression */
      char resvd;               /* reserved--set to 0 */
      short minimumHeight;      /* minimum height */
      short minimumWidth;       /* minimum width */
      short decompressPipelineLatency;
                                /* in milliseconds (asynchronous) */
      short compressPipelineLatency;
                                /* in milliseconds (asynchronous) */
      long privateData;         /* reserved for use by Apple */
};
typedef struct CodecInfo CodecInfo;

/* compressor name structure returned by GetCodecNameList function */
struct CodecNameSpec
{
      CodecComponent codec;/* component ID for compressor */
      CodecType cType;     /* type identifier for compressor */
      Str31 typeName;      /* string identifier of compression algorithm */
      Handle name;         /* name of compressor component */
};
typedef struct CodecNameSpec CodecNameSpec;

/* compressor name list structure */
struct CodecNameSpecList
{
      short count;            /* number of compressor name structures in list
                                 array that follows */
      CodecNameSpec list[1];  /* array of compressor name structures */
};
typedef struct CodecNameSpecList CodecNameSpecList;
typedef CodecNameSpecList *CodecNameSpecListPtr;

/*
   flags from message parameter of application-defined progress functions
   tell why the Image Compression Manager called your function
*/
enum
```

```
{
   codecProgressOpen          = 0,  /* start of a long operation */
   codecProgressUpdatePercent = 1,  /* passes completion information */
   codecProgressClose         = 2   /* end of a long operation*/
};
typedef pascal void (*CompletionProcPtr) (OSErr result, short flags,
                     long refCon);

/* data rate parameters structure */
typedef struct {
   long     dataRate;               /* bytes per second */
   long     dataOverrun;            /* number of bytes outside rate */
   long     frameDuration;          /* in milliseconds */
   long     keyFrameRate;           /* frequency of key frames */
   CodecQ   minSpatialQuality;      /* minimum spatial quality */
   CodecQ   minTemporalQuality;     /* minimum temporal quality */
} DataRateParams;
typedef DataRateParams *DataRateParamsPtr;
```

Image Compression Manager Functions

Getting Information About Compressor Components

```
pascal OSErr CodecManagerVersion
                            (long *version);
pascal OSErr GetCodecNameList
                            (CodecNameSpecListPtr *list, short showAll);
pascal OSErr DisposeCodecNameList
                            (CodecNameSpecListPtr list);
pascal OSErr GetCodecInfo   (CodecInfo *info, CodecType cType,
                             CodecComponent codec);
pascal OSErr FindCodec      (CodecType cType, CodecComponent specCodec,
                             CompressorComponent *compressor,
                             DecompressorComponent *decompressor);
```

Getting Information About Compressed Data

```
pascal OSErr GetMaxCompressionSize
                            (PixMapHandle src, const Rect *srcRect,
                             short colorDepth, CodecQ quality,
                             CodecType cType, CompressorComponent codec,
                             long *size);
```

```
pascal OSErr GetCompressionTime
                            (PixMapHandle src, const Rect *srcRect,
                             short colorDepth, CodecType cType,
                             CompressorComponent codec,
                             CodecQ *spatialQuality,
                             CodecQ *temporalQuality,
                             unsigned long *compressTime);
pascal OSErr GetSimilarity  (PixMapHandle src, const Rect *srcRect,
                             ImageDescriptionHandle desc, Ptr data,
                             Fixed *similarity);
pascal OSErr GetCompressedImageSize
                            (ImageDescriptionHandle desc, Ptr data,
                             long bufferSize, DataProcRecordPtr dataProc,
                             long *dataSize);
```

Working With Images

```
pascal OSErr CompressImage  (PixMapHandle src, const  Rect *srcRect,
                             CodecQ quality, CodecType cType,
                             ImageDescriptionHandle desc, Ptr data);
pascal OSErr FCompressImage
                            (PixMapHandle src, const  Rect *srcRect,
                             short colorDepth, CodecQ quality,
                             CodecType cType, CompressorComponent codec,
                             CTabHandle clut, CodecFlags flags,
                             long bufferSize, FlushProcRecordPtr flushProc,
                             ProgressProcRecordPtr progressProc,
                             ImageDescriptionHandle desc, Ptr data);
pascal OSErr DecompressImage
                            (Ptr data, ImageDescriptionHandle desc,
                             PixMapHandle dst, const Rect *srcRect,
                             const  Rect *dstRect, short mode,
                             RgnHandle mask);
pascal OSErr FDecompressImage
                            (Ptr data, ImageDescriptionHandle desc,
                             PixMapHandle dst, const  Rect *srcRect,
                             MatrixRecordPtr matrix, short mode,
                             RgnHandle mask, PixMapHandle matte,
                             const  Rect *matteRect, CodecQ accuracy,
                             DecompressorComponent codec, long bufferSize,
                             DataProcRecordPtr dataProc,
                             ProgressProcRecordPtr progressProc);
```

```
pascal OSErr ConvertImage   (ImageDescriptionHandle srcDD, Ptr srcData,
                             short colorDepth, CTabHandle clut,
                             CodecQ accuracy, CodecQ quality,
                             CodecType cType, CodecComponent codec,
                             ImageDescriptionHandle dstDD, Ptr dstData);
pascal OSErr TrimImage      (ImageDescriptionHandle desc, Ptr inData,
                             long inBufferSize, DataProcRecordPtr dataProc,
                             Ptr outData, long outBufferSize,
                             FlushProcRecordPtr flushProc, Rect *trimRect,
                             ProgressProcRecordPtr progressProc);
pascal OSErr SetImageDescriptionCTable
                            (ImageDescriptionHandle desc,
                             CTabHandle ctable);
pascal OSErr GetImageDescriptionCTable
                            (ImageDescriptionHandle desc,
                             CTabHandle *ctable);
```

Working With Pictures and PICT Files

```
pascal OSErr CompressPicture
                            (PicHandle srcPicture, PicHandle dstPicture,
                             CodecQ quality, CodecType cType);
pascal OSErr FCompressPicture
                            (PicHandle srcPicture, PicHandle dstPicture,
                             short colorDepth, CTabHandle clut,
                             CodecQ quality, short doDither,
                             short compressAgain,
                             ProgressProcRecordPtr progressProc,
                             CodecType cType, CompressorComponent codec);
pascal OSErr CompressPictureFile
                            (short srcRefNum, short dstRefNum,
                             CodecQ quality, CodecType cType);
pascal OSErr FCompressPictureFile
                            (short srcRefNum, short dstRefNum,
                             short colorDepth, CTabHandle clut,
                             CodecQ quality, short doDither,
                             short compressAgain,
                             ProgressProcRecordPtr progressProc,
                             CodecType cType, CompressorComponent codec);
pascal OSErr DrawPictureFile
                            (short refNum, const Rect *frame,
                             ProgressProcRecordPtr progressProc);
```

```
pascal OSErr DrawTrimmedPicture
                            (PicHandle srcPicture, const Rect *frame,
                             RgnHandle trimMask, short doDither,
                             ProgressProcRecordPtr progressProc);
pascal OSErr DrawTrimmedPictureFile
                            (short srcRefnum, const Rect *frame,
                             RgnHandle trimMask, short doDither,
                             ProgressProcRecordPtr progressProc);
pascal OSErr GetPictureFileHeader
                            (short refNum, Rect *frame,
                             OpenCPicParams *header);
```

Making Thumbnail Pictures

```
pascal OSErr MakeThumbnailFromPicture
                            (PicHandle picture, short colorDepth,
                             PicHandle thumbnail,
                             ProgressProcRecordPtr progressProc);
pascal OSErr MakeThumbnailFromPictureFile
                            (short refNum, short colorDepth,
                             PicHandle thumbnail,
                             ProgressProcRecordPtr progressProc);
pascal OSErr MakeThumbnailFromPixMap
                            (PixMapHandle src, const Rect *srcRect,
                             short colorDepth, PicHandle thumbnail,
                             ProgressProcRecordPtr progressProc);
```

Working With Sequences

```
pascal OSErr CompressSequenceBegin
                            (ImageSequence *seqID, PixMapHandle src,
                             PixMapHandle prev, const Rect *srcRect,
                             const  Rect *prevRect, short colorDepth,
                             CodecType cType, CompressorComponent codec,
                             CodecQ spatialQuality, CodecQ temporalQuality,
                             long keyFrameRate, CTabHandle clut,
                             CodecFlags flags, ImageDescriptionHandle desc);
pascal OSErr CompressSequenceFrame
                            (ImageSequence seqID, PixMapHandle src,
                             const Rect *srcRect, CodecFlags flags,
                             Ptr data, long *dataSize,
                             unsigned char *similarity,
                             CompletionProcRecordPtr asyncCompletionProc);
```

```
pascal OSErr DecompressSequenceBegin
                            (ImageSequence *seqID,
                             ImageDescriptionHandle desc, CGrafPtr port,
                             GDHandle gdh, const Rect *srcRect,
                             MatrixRecordPtr matrix, short mode,
                             RgnHandle mask, CodecFlags flags,
                             CodecQ accuracy, DecompressorComponent codec);
pascal OSErr DecompressSequenceFrame
                            (ImageSequence seqID, Ptr data,
                             CodecFlags inFlags, CodecFlags *outFlags,
                             CompletionProcRecordPtr asyncCompletionProc);
pascal OSErr CDSequenceBusy
                            (ImageSequence seqID);
pascal OSErr CDSequenceEnd  (ImageSequence seqID);
```

Changing Sequence-Compression Parameters

```
pascal OSErr SetCSequenceQuality
                            (ImageSequence seqID, CodecQ spatialQuality,
                             CodecQ temporalQuality);
pascal OSErr SetCSequenceKeyFrameRate
                            (ImageSequence seqID, long keyframerate);
pascal OSErr GetCSequenceKeyFrameRate
                            (ImageSequence seqID, long *keyframerate);
pascal OSErr SetCSequenceFrameNumber
                            (ImageSequence seqID, long frameNumber);
pascal OSErr GetCSequenceFrameNumber
                            (ImageSequence seqID, long *frameNumber);
pascal OSErr SetCSequencePrev
                            (ImageSequence seqID, PixMapHandle prev,
                             const Rect *prevRect);
pascal OSErr SetCSequenceFlushProc
                            (ImageSequence seqID,
                             FlushProcRecordPtr flushProc, long bufferSize);
pascal OSErr GetCSequencePrevBuffer
                            (ImageSequence seqID, GWorldPtr *gworld);
```

Constraining Compressed Data

```
pascal OSErr SetCSequenceDataRateParams
                            (ImageSequence seqID,
                             DataRateParamsPtr params);
pascal OSErr GetCSequenceDataRateParams
                            (ImageSequence seqID, DataRateParamsPtr params);
```

Changing Sequence-Decompression Parameters

```
pascal OSErr SetDSequenceTransferMode
                            (ImageSequence seqID, short mode,
                             const RGBColor *opColor);
pascal OSErr SetDSequenceSrcRect
                            (ImageSequence seqID, const Rect *srcRect);
pascal OSErr SetDSequenceMatrix
                            (ImageSequence seqID, MatrixRecordPtr matrix);
pascal OSErr SetDSequenceMask
                            (ImageSequence seqID, RgnHandle mask);
pascal OSErr SetDSequenceMatte
                            (ImageSequence seqID, PixMapHandle matte,
                             const Rect *matteRect);
pascal OSErr SetDSequenceAccuracy
                            (ImageSequence seqID, CodecQ accuracy);
pascal OSErr SetDSequenceDataProc
                            (ImageSequence seqID,
                             DataProcRecordPtr dataProc, long bufferSize);
pascal OSErr GetDSequenceImageBuffer
                            (ImageSequence seqID, GWorldPtr *gworld);
pascal OSErr GetDSequenceScreenBuffer
                            (ImageSequence seqID, GWorldPtr *gworld);
```

Working With the StdPix Function

```
pascal void StdPix          (PixMapPtr src, const  Rect *srcRect,
                             MatrixRecordPtr matrix, short mode,
                             RgnHandle mask, PixMapPtr matte,
                             Rect *matteRect, short flags);
pascal OSErr SetCompressedPixMapInfo
                            (PixMapPtr pix, ImageDescriptionHandle desc,
                             Ptr data, long bufferSize,
                             DataProcRecordPtr dataProc,
                             ProgressProcRecordPtr progressProc);
pascal OSErr GetCompressedPixMapInfo
                            (PixMapPtr pix, ImageDescriptionHandle *desc,
                             Ptr *data, long *bufferSize,
                             DataProcRecord *dataProc,
                             ProgressProcRecord *progressProc);
```

Aligning Windows

```
pascal void AlignWindow     (WindowPtr wp, Boolean front,
                             const Rect *alignmentRect,
                             AlignmentProcRecordPtr alignmentProc);
pascal void DragAlignedWindow
                            (WindowPtr wp, Point startPt,
                             Rect *boundsRect, Rect *alignmentRect,
                             AlignmentProcRecordPtr alignmentProc);
pascal long DragAlignedGrayRgn
                            (RgnHandle theRgn, Point startPt,
                             Rect *boundsRect, Rect *slopRect, short axis,
                             ProcPtr actionProc, Rect *alignmentRect,
                             AlignmentProcRecordPtr alignmentProc);
pascal void AlignScreenRect
                            (Rect *rp,
                             AlignmentProcRecordPtr alignmentProc);
```

Working With Graphics Devices and Graphics Worlds

```
pascal OSErr GetBestDeviceRect
                            (GDHandle *gdh, Rect *rp);
pascal QDErr NewImageGWorld
                            (GWorldPtr *gworld,
                             ImageDescription **idh,GWorldFlags flags);
```

Application-Defined Functions

Data-Loading Functions

```
pascal OSErr MyDataLoadingProc
                            (Ptr *dataP, long bytesNeeded, long refcon);
```

Data-Unloading Functions

```
pascal OSErr MyDataUnloadingProc
                            (Ptr data, long bytesAdded, long refcon);
```

Progress Functions

```
pascal  OSErr MyProgressProc
                            (short message, Fixed completeness,
                             long refcon);
```

Completion Functions

```
pascal OSErr MyCompletionProc
                            (OSErr result, short flag, long refcon);
```

Alignment Functions

```
pascal void MyAlignmentProc
                            (Rect *rp, long refcon);
```

Pascal Summary

Constants

```
CONST
   gestaltCompressionMgr            = 'icmp';{determines if Image }
                                             { Compression Manager is }
                                             { available}

   codecMinimumDataSize             = 32768; {smallest data buffer you may }
                                             { allow for data spooling}
   compressorComponentType          = 'imco';{compressor component type}
   decompressorComponentType        = 'imdc';{decompressor component type}

   {Image Compression Manager function control flags}
   codecFlagUseImageBuffer          = $1;    {(input) use offscreen buffer}
   codecFlagUseScreenBuffer         = $2;    {(input) use screen buffer}
   codecFlagUpdatePrevious          = $4;    {(input) previous image is }
                                             { updated}
   codecFlagNoScreenUpdate          = $8;    {(input) no screen image update}
   codecFlagWasCompressed           = $10;   {(input) image has been }
                                             { compressed}
   codecFlagDontOffscreen           = $20;   {don't use offscreen buffer}
   codecFlagUpdatePreviousComp      = $40;   {(input) previous image buffer }
                                             { updated}
   codecFlagForceKeyFrame           = $80;   {force key frame from image}
   codecFlagOnlyScreenUpdate        = $100;  {decompresses current frame}
   codecFlagLiveGrab                = $200;  {sequence from live video grab}
```

```
   codecFlagDontUseNewImageBuffer   = $400;  {(input) return error if image }
                                             { buffer is new or reallocated}
   codecFlagInterlaceUpdate         = $800;  {(input) use interlaced update}

   {status flags from outflags parameter of DecompressSequenceFrame function}
   codecFlagUsedNewImageBuffer      = $4000; {(output) used new image buffer}
   codecFlagUsedImageBuffer         = $8000; {(output) used offscreen image }
                                             { buffer}

   {completion flags from application-defined completion functions}
   codecCompletionSource            = 1;     {done with source buffer}
   codecCompletionDest              = 2;     {done with destination buffer}

   {flags from application-defined progress functions message parameter—- }
   { tell why the Image Compression Manager called your function}
   codecProgressOpen                = 0;     {start of a long operation}
   codecProgressUpdatePercent       = 1;     {passing completion data}
   codecProgressClose               = 2;     {end of a long}

   {compression quality values}
   codecMinQuality         = $000;  {minimum-quality image reproduction}
   codecLowQuality         = $100;  {low-quality image reproduction}
   codecNormalQuality      = $200;  {normal-quality image reproduction}
   codecHighQuality        = $300;  {high-quality image reproduction}
   codecMaxQuality         = $3FF;  {maximum-quality image reproduction}
   codecLosslessQuality    = $400;  {lossless-quality image reproduction}

   {special compressor and decompressor identifiers}
   anyCodec                =  0; {first component of specified type}
   bestSpeedCodec          = -1; {fastest component of specified type}
   bestFidelityCodec       = -2; {most accurate component of specified type}
   bestCompressionCodec    = -3; {component with smallest resulting data}

{constants for doDither parameter of DrawTrimmedPictureFile and }
{ FCompressPictureFile functions}

   defaultDither        = 0;  {respect dithering instructions in source }
                              { picture}
   forceDither          = 1;  {dither image}
   suppressDither       = 2;  {don't dither image}
```

Data Types

```
TYPE
   CompressorComponent              = Component;{compressor identifier}
   DecompressorComponent            = Component;{decompressor identifier}
   CodecComponent                   = Component;{compressor identifier}

   CodecType                        = OSType;   {compressor type}

   CodecFlags                       = Integer;  {compressor component flags}

   CodecQ                           = LongInt;  {compression quality}

   DataProcPtr             = ProcPtr;  {pointer to a data-loading function}
   FlushProcPtr            = ProcPtr;  {pointer to a data-unloading function}
   CompletionProcPtr       = ProcPtr;  {pointer to a completion function}
   ProgressProcPtr         = ProcPtr;  {pointer to a progress function}

   ImageSequence           = LongInt;  {unique sequence identifer}

   ProgressProcRecordPtr   = ^ProgressProcRecord;
   ProgressProcRecord =                    {progress function record}
   RECORD
      progressProc:     ProgressProcPtr;  {pointer to your progress function}
      progressRefCon:   LongInt;          {reference constant}
   END;

   CompletionProcRecordPtr = ^CompletionProcRecord;
   CompletionProcRecord =                  {completion function record}
   RECORD
      completionProc:   CompletionProcPtr;{pointer to completion function}
      completionRefCon: LongInt;          {reference constant}
   END;

   DataProcRecordPtr             = ^DataProcRecord;
   DataProcRecord =                        {data-loading function record}
   RECORD
      dataProc:   DataProcPtr;            {pointer to data-loading function}
      dataRefCon: LongInt;                {reference constant}
   END;

   FlushProcRecordPtr            = ^FlushProcRecord;
   FlushProcRecord =                       {data-unloading function record}
```

```
RECORD
   flushProc:     FlushProcPtr;        {pointer to data-unloading }
                                       { function}
   flushRefCon:   LongInt;             {reference constant}
END;

ImageDescriptionPtr                 = ^ImageDescription;
ImageDescriptionHandle              = ^ImageDescriptionPtr;
ImageDescription =
PACKED RECORD
   idSize:           LongInt;          {total size of this record}
   cType:            CodecType;        {type of creator component}
   resvd1:           LongInt;          {reserved—-must be set to 0}
   resvd2:           Integer;          {reserved—-must be set to 0}
   dataRefIndex:     Integer;          {reserved—-must be set to 0}
   version:          Integer;          {version of compressed data}
   revisionLevel:    Integer;          {version of creator compressor}
   vendor:           LongInt;          {developer of creator compressor}
   temporalQuality:  CodecQ;           {degree of temporal compression}
   spatialQuality:   CodecQ;           {degree of spatial compression}
   width:            Integer;    {width of source image in pixels}
   height:           Integer;    {height of source image in pixels}
   hRes:             Fixed;      {horizontal resolution of source image}
   vRes:             Fixed;      {vertical resolution of source image}
   dataSize:         LongInt;    {byte size of compressed image data}
   frameCount:       Integer;    {number of frames in image data}
   name:             PACKED ARRAY[0..31] of char;
                               {name of compression algorithm}
   depth:            Integer;    {pixel depth of source image}
   clutID:           Integer; {ID number of the color table for image}
END;

CodecInfo =                          {compressor information record}
PACKED RECORD
   typeName:            PACKED ARRAY[0..31] of char;
                                    {compression algorithm}
   version:             Integer;    {version of compressed data}
   revisionLevel:       Integer;    {version of component}
   vendor:              LongInt;    {developer of component}
   decompressFlags:     LongInt;    {decompression capability flags}
   compressFlags:       LongInt;    {compression capability flags}
   formatFlags:         LongInt;    {format flags}
   compressionAccuracy:
                        Char;       {relative accuracy of compression}
```

```
   decompressionAccuracy:
                        Char;       {relative accuracy of decompression}
   compressionSpeed:    Integer;    {relative compression speed}
   decompressionSpeed:  Integer;    {relative decompression speed}
   compressionLevel:    Char;       {relative compression of component}
   resvd:               Char;       {reserved—-set to 0}
   minimumHeight:       Integer;    {minimum height in pixels}
   minimumWidth:        Integer;    {maximum width in pixels}
   decompressPipelineLatency:
                        Integer;    {milliseconds (asynchronous)}
   compressPipelineLatency:
                        Integer;    {milliseconds (asynchronous)}
   privateData:         LongInt;    {reserved--must be set to 0}
END;

{compressor name record returned by GetCodecNameList}
CodecNameSpec =
PACKED RECORD
   codec:      CodecComponent;{component ID for compressor}
   cType:      CodecType;     {type identifier for compressor}
   typeName:   PACKED ARRAY[0..31] OF Char;
                              {string identifier of compression algorithm}
   name:       Handle;        {name of compressor component}
END;

{compressor name list record}
CodecNameSpecListPtr             = ^CodecNameSpecList;
CodecNameSpecList =
RECORD
   count:      Integer;             {number of compressor name records}
   list:       ARRAY[0..0] OF CodecNameSpec;
                                    {array of compressor name records}
END;

{data rate parameters record}
DataRateParamsPtr = ^DataRateParams;
DataRateParams =
RECORD
   dataRate:            LongInt;    {bytes per second}
   dataOverrun:         LongInt;    {number of bytes outside rate}
   frameDuration:       LongInt;    {duration in milliseconds}
   keyFrameRate:        LongInt;    {frequency of key frames}
```

```
    minSpatialQuality:   CodecQ;     {minimum spatial quality}
    minTemporalQuality:  CodecQ;     {minimum temporal quality}
  END;
```

Image Compression Manager Routines

Getting Information About Compressor Components

```
FUNCTION CodecManagerVersion
                          (VAR version: LongInt): OSErr;
FUNCTION GetCodecNameList (VAR list: CodecNameSpecListPtr;
                           showAll: Integer): OSErr;
FUNCTION DisposeCodecNameList
                          (list: CodecNameSpecListPtr): OSErr;
FUNCTION GetCodecInfo     (VAR info: CodecInfo; cType: CodecType;
                           codec: CodecComponent): OSErr;
FUNCTION FindCodec        (cType: CodecType; specCodec: CodecComponent;
                           VAR compressor: CompressorComponent;
                           VAR decompressor: DecompressorComponent):
                           OSErr;
```

Getting Information About Compressed Data

```
FUNCTION GetMaxCompressionSize
                          (src: PixMapHandle; srcRect: Rect;
                           colorDepth: Integer; quality: CodecQ;
                           cType: CodecType; codec: CompressorComponent;
                           VAR size: LongInt): OSErr;
FUNCTION GetCompressionTime
                          (src: PixMapHandle; srcRect: Rect;
                           colorDepth: Integer; cType: CodecType;
                           codec: CompressorComponent;
                           VAR spatialQuality: CodecQ;
                           VAR temporalQuality: CodecQ;
                           VAR compressTime: LongInt): OSErr;
FUNCTION GetSimilarity    (src: PixMapHandle; srcRect: Rect;
                           desc: ImageDescriptionHandle; data: Ptr;
                           VAR similarity: Fixed): OSErr;
FUNCTION GetCompressedImageSize
                          (desc: ImageDescriptionHandle; data: Ptr;
                           bufferSize: LongInt;
                           dataProc: DataProcRecordPtr;
                           VAR dataSize: LongInt): OSErr;
```

Working With Images

```
FUNCTION CompressImage      (src: PixMapHandle; srcRect: Rect;
                             quality: CodecQ; cType: CodecType;
                             desc: ImageDescriptionHandle;
                             data: Ptr): OSErr;
FUNCTION FCompressImage     (src: PixMapHandle; srcRect: Rect;
                             colorDepth: Integer; quality: CodecQ;
                             cType: CodecType; codec: CompressorComponent;
                             clut: CTabHandle; flags: CodecFlags;
                             bufferSize: LongInt;
                             flushProc: FlushProcRecordPtr;
                             progressProc: ProgressProcRecordPtr;
                             desc: ImageDescriptionHandle;
                             data: Ptr): OSErr;
FUNCTION DecompressImage    (data: Ptr; desc: ImageDescriptionHandle;
                             dst: PixMapHandle; srcRect: Rect;
                             dstRect: Rect; mode: Integer;
                             mask: RgnHandle): OSErr;
FUNCTION FDecompressImage   (data: Ptr; desc: ImageDescriptionHandle;
                             dst: PixMapHandle; srcRect: Rect;
                             matrix: MatrixRecordPtr; mode: Integer;
                             mask: RgnHandle; matte: PixMapHandle;
                             matteRect: Rect; accuracy: CodecQ;
                             codec: DecompressorComponent;
                             bufferSize: LongInt;
                             dataProc: DataProcRecordPtr;
                             progressProc: ProgressProcRecordPtr): OSErr;
FUNCTION ConvertImage       (srcDD: ImageDescriptionHandle; srcData: Ptr;
                             colorDepth: Integer; clut: CTabHandle;
                             accuracy: CodecQ; quality: CodecQ;
                             cType: CodecType; codec: CodecComponent;
                             dstDD: ImageDescriptionHandle;
                             dstData: Ptr): OSErr;
FUNCTION TrimImage          (desc: ImageDescriptionHandle; inData: Ptr;
                             inBufferSize: LongInt;
                             dataProc: DataProcRecordPtr; outData: Ptr;
                             outBufferSize: LongInt;
                             flushProc: FlushProcRecordPtr;
                             VAR trimRect: Rect;
                             progressProc: ProgressProcRecordPtr): OSErr;
FUNCTION SetImageDescriptionCTable
                            (desc: ImageDescriptionHandle;
                             ctable: CTabHandle): OSErr;
```

```
FUNCTION GetImageDescriptionCTable
                            (desc: ImageDescriptionHandle;
                             VAR ctable: CTabHandle): OSErr;
```

Working With Pictures and PICT Files

```
FUNCTION CompressPicture    (srcPicture: PicHandle; dstPicture: PicHandle;
                             quality: CodecQ; cType: CodecType): OSErr;
FUNCTION FCompressPicture   (srcPicture: PicHandle; dstPicture: PicHandle;
                             colorDepth: Integer; clut: CTabHandle;
                             quality: CodecQ; doDither: Integer;
                             compressAgain: Integer;
                             progressProc: ProgressProcRecordPtr;
                             cType: CodecType;
                             codec: CompressorComponent): OSErr;
FUNCTION CompressPictureFile
                            (srcRefNum: Integer; dstRefNum: Integer;
                             quality: CodecQ; cType: CodecType): OSErr;
FUNCTION FCompressPictureFile
                            (srcRefNum: Integer; dstRefNum: Integer;
                             colorDepth: Integer; clut: CTabHandle;
                             quality: CodecQ; doDither: Integer;
                             compressAgain: Integer;
                             progressProc: ProgressProcRecordPtr;
                             cType: CodecType;
                             odec: CompressorComponent): OSErr;
FUNCTION DrawPictureFile    (refNum: Integer; frame: Rect;
                             progressProc: ProgressProcRecordPtr): OSErr;
FUNCTION DrawTrimmedPicture
                            (srcPicture: PicHandle; frame: Rect;
                             trimMask: RgnHandle; doDither: Integer;
                             progressProc: ProgressProcPtr): OSErr;
FUNCTION DrawTrimmedPictureFile
                            (srcRefnum: Integer; frame: Rect;
                             trimMask: RgnHandle; doDither: Integer;
                             progressProc: ProgressProcRecordPtr): OSErr;
FUNCTION GetPictureFileHeader
                            (refNum: Integer; VAR frame: Rect;
                             VAR header: OpenCPicParams): OSErr;
```

Making Thumbnail Pictures

```
FUNCTION MakeThumbnailFromPicture
                            (picture: PicHandle; colorDepth: Integer;
                             thumbnail: PicHandle;
                             progressProc: ProgressProcRecordPtr): OSErr;
```

```
FUNCTION MakeThumbnailFromPictureFile
                            (refNum: Integer; colorDepth: Integer;
                             thumbnail: PicHandle;
                             progressProc: ProgressProcRecordPtr): OSErr;
FUNCTION MakeThumbnailFromPixMap
                            (src: PixMapHandle; srcRect: Rect;
                             colorDepth: Integer; thumbnail: PicHandle;
                             progressProc: ProgressProcRecordPtr): OSErr;
```

Working With Sequences

```
FUNCTION CompressSequenceBegin
                            (VAR seqID: ImageSequence;
                             src: PixMapHandle; prev: PixMapHandle;
                             srcRect: Rect; prevRect: Rect;
                             colorDepth: Integer; cType: CodecType;
                             codec: CompressorComponent;
                             spatialQuality: CodecQ;
                             temporalQuality: CodecQ;
                             keyFrameRate: LongInt; clut: CTabHandle;
                             flags: CodecFlags;
                             desc: ImageDescriptionHandle): OSErr;
FUNCTION CompressSequenceFrame
                            (seqID: ImageSequence;
                             src: PixMapHandle; srcRect: Rect;
                             flags: CodecFlags; data: Ptr;
                             VAR dataSize: LongInt; VAR similarity: Char;
                             asyncCompletionProc: CompletionProcRecordPtr):
                             OSErr;
FUNCTION DecompressSequenceBegin
                            (VAR seqID: ImageSequence;
                             desc: ImageDescriptionHandle; port: CGrafPtr;
                             gdh: GDHandle; srcRect: Rect;
                             matrix: MatrixRecordPtr; mode: Integer;
                             mask: RgnHandle; flags: CodecFlags;
                             accuracy: CodecQ;
                             codec: DecompressorComponent): OSErr;
FUNCTION DecompressSequenceFrame
                            (seqID: ImageSequence; data: Ptr;
                             inFlags: CodecFlags; VAR outFlags: CodecFlags;
                             asyncCompletionProc: CompletionProcRecordPtr):
                             OSErr;
FUNCTION CDSequenceBusy     (seqID: ImageSequence): OSErr;
FUNCTION CDSequenceEnd      (seqID: ImageSequence): OSErr;
```

Changing Sequence-Compression Parameters

```
FUNCTION SetCSequenceQuality
                            (seqID: ImageSequence; spatialQuality: CodecQ;
                             temporalQuality: CodecQ): OSErr;
FUNCTION SetCSequenceKeyFrameRate
                            (seqID: ImageSequence;
                             keyframerate: LongInt): OSErr;
FUNCTION GetCSequenceKeyFrameRate
                            (seqID: ImageSequence;
                             VAR keyframerate: LongInt): OSErr;
FUNCTION GetCSequenceKeyFrameRate
                            (seqID: ImageSequence;
                             VAR keyframerate: LongInt): OSErr;
FUNCTION SetCSequenceFrameNumber
                            (seqID: ImageSequence;
                             frameNumber: LongInt): OSErr;
FUNCTION GetCSequenceFrameNumber
                            (seqID: ImageSequence;
                             VAR frameNumber: LongInt): OSErr;
FUNCTION SetCSequencePrev   (seqID: ImageSequence; prev: PixMapHandle;
                             prevRect: Rect): OSErr;
FUNCTION SetCSequenceFlushProc
                            (seqID: ImageSequence;
                             flushProc: FlushProcRecordPtr;
                             bufferSize: LongInt): OSErr;
FUNCTION GetCSequencePrevBuffer
                            (seqID: ImageSequence;
                             VAR gworld: GWorldPtr): OSErr;
```

Constraining Compressed Data

```
FUNCTION SetCSequenceDataRateParams
                            (seqID: ImageSequence;
                             params: DataRateParamsPtr): OSErr;
FUNCTION GetCSequenceDataRateParams
                            (seqID: ImageSequence;
                             params: DataRateParamsPtr): OSErr;
```

Changing Sequence-Decompression Parameters

```
FUNCTION SetDSequenceTransferMode
                            (seqID: ImageSequence; mode: Integer;
                             opColor: RGBColor): OSErr;
```

```
FUNCTION SetDSequenceSrcRect
                            (seqID: ImageSequence; srcRect: Rect): OSErr;
FUNCTION SetDSequenceMatrix
                            (seqID: ImageSequence;
                             matrix: MatrixRecordPtr): OSErr;
FUNCTION SetDSequenceMask   (seqID: ImageSequence; mask: RgnHandle): OSErr;
FUNCTION SetDSequenceMatte  (seqID: ImageSequence; matte: PixMapHandle;
                             matteRect: Rect): OSErr;
FUNCTION SetDSequenceAccuracy
                            (seqID: ImageSequence;
                             accuracy: CodecQ): OSErr;
FUNCTION SetDSequenceDataProc
                            (seqID: ImageSequence;
                             dataProc: DataProcRecordPtr;
                             bufferSize: LongInt): OSErr;
FUNCTION GetDSequenceImageBuffer
                            (seqID: ImageSequence;
                             VAR gworld: GWorldPtr): OSErr;
FUNCTION GetDSequenceScreenBuffer
                            (seqID: ImageSequence;
                             VAR gworld: GWorldPtr): OSErr;
```

Working With the StdPix Routine

```
FUNCTION StdPix             (src: PixMapPtr; srcRect: Rect;
                             matrix: MatrixRecordPtr; mode: Integer;
                             mask: RgnHandle; matte: PixMapPtr;
                             matteRect: Rect; flags: Integer): OSErr
FUNCTION SetCompressedPixMapInfo
                            (pix: PixMapPtr; desc: ImageDescriptionHandle;
                             data: Ptr; bufferSize: LongInt;
                             dataProc: DataProcRecordPtr;
                             progressProc: ProgressProcRecordPtr): OSErr;
FUNCTION GetCompressedPixMapInfo
                            (pix: PixMapPtr;
                             VAR desc: ImageDescriptionHandle;
                             VAR data: Ptr; VAR bufferSize: LongInt;
                             VAR dataProc: DataProcRecord;
                             VAR progressProc: ProgressProcRecord): OSErr;
```

Aligning Windows

```
PROCEDURE AlignWindow       (wp: WindowPtr; front: Boolean;
                             alignmentRect: RectPtr;
                             alignmentProc:AlignmentProcRecordPtr );
```

```
PROCEDURE DragAlignedWindow
                            (wp: WindowPtr; startPt: Point;
                             VAR boundsRect: Rect; VAR alignmentRect: Rect;
                             alignmentProc:AlignmentProcRecordPtr);
FUNCTION DragAlignedGrayRgn
                            (theRgn: RgnHandle; startPt: Point;
                             VAR boundsRect: Rect; VAR slopRect: Rect;
                             axis: Integer; actionProc: ProcPtr;
                             VAR alignmentRect: Rect;
                             alignmentProc:AlignmentProcRecordPtr): LongInt;
PROCEDURE AlignScreenRect   (VAR rp: Rect;
                             alignmentProc:AlignmentProcRecordPtr);
```

Working With Graphics Devices and Graphics Worlds

```
FUNCTION GetBestDeviceRect  (VAR gdh: GDHandle; VAR rp: Rect): OSErr;
FUNCTION NewImageGWorld     (VAR gworld: GWorldPtr;
                             idh: ImageDescriptionHandle;
                             flags :GWorldFlags): OSErr;
```

Application-Defined Routines

Data-Loading Functions

```
FUNCTION MyDataLoadingProc  (VAR dataP: Ptr; bytesNeeded: LongInt;
                             refcon: LongInt): OSErr;
```

Data-Unloading Functions

```
FUNCTION MyDataUnloadingProc
                            (data: Ptr; bytesAdded: LongInt;
                             refcon: LongInt): OSErr;
```

Progress Functions

```
FUNCTION MyProgressProc     (message: Integer; completeness: Fixed;
                             refcon: LongInt): OSErr;
```

Completion Functions

```
FUNCTION MyCompletionProc   (result: OSErr; flag: Integer;
                             refcon: LongInt): OSErr;
```

Alignment Routines

```
PROCEDURE MyAlignmentProc   (rp: RectPtr, refcon: LongInt);
```

Result Codes

paramErr	–50	Invalid parameter specified
memFullErr	–108	Not enough memory available
codecErr	–8960	General error condition
noCodecErr	–8961	Image Compression Manager could not find the specified compressor
codecUnimpErr	–8962	Feature not implemented by this compressor
codecSizeErr	–8963	Invalid buffer size specified
codecScreenBufErr	–8964	Could not allocate the screen buffer
codecImageBufErr	–8965	Could not allocate the image buffer
codecSpoolErr	–8966	Error loading or unloading data
codecAbortErr	–8967	Operation aborted by the progress function
codecWouldOffScreenErr	–8968	Compressor would use screen buffer if it could
codecBadDataErr	–8969	Compressed data contains inconsistencies
codecDataVersErr	–8970	Compressor does not support the compression version used to compress the image
codecExtensionNotFoundErr	–8971	Requested extension is not in the image description
codecConditionErr	–8972	Component cannot perform requested operation
codecOpenErr	–8973	Could not open the compressor or decompressor

CHAPTER 4

Movie Resource Formats

Contents

This chapter describes the format of QuickTime movie resources. **Movie resources** are the data structures that provide the medium of exchange for movie data. Movie resources may be exchanged between applications on a single Macintosh computer or between applications on several Macintosh and non-Macintosh computers.

IMPORTANT
The information in this chapter is intended for developers who need to know about the content of movie resources. You need to learn about movie resources if you want to create movies on other computer environments and import them to the Macintosh environment, or if you want to interpret QuickTime movies on other types of computers. Developers of Macintosh applications do not need to know about the layout of movie resources—the Movie Toolbox functions handle the details of movie storage and exchange. s

This chapter describes **atoms,** the basic storage elements that, taken together, make up a movie resource.

The chapter is divided into the following major sections:

- "Storing Movies in Files" describes the two ways that QuickTime movies may be stored in files.
- "Atoms" describes the format and content of the most basic movie storage element and the standard atoms that may be found in a movie resource.
- "Overview of the Movie Resource Atom" provides a look at the movie resource structure of a QuickTime movie.
- "Using Media Information Atoms" provides examples of the media information atoms.

To understand fully the information presented in this chapter, you should be familiar with the Movie Toolbox (see the chapter "Movie Toolbox" in this book). In particular, you should read about the characteristics of movie, track, and media structures.

If you are developing a movie application that runs on another type of computer, you do not have the facilities of the Movie Toolbox available to you. If you want that application to exchange data with QuickTime applications on the Macintosh computer, you need to know the format of QuickTime movie resources.

Introduction to Movie Resources

Movie resources define the data interchange format for movies. The Movie Toolbox allows your application to create, view, edit, and store QuickTime movies. The functions of the Movie Toolbox shield your program from the details of how a movie is stored in the Macintosh file system (in the file type `'MooV'`). As a result, applications that run on Macintosh computers do not need to know anything about the internal structures of movie resources or movie files.

Storing Movies in Files

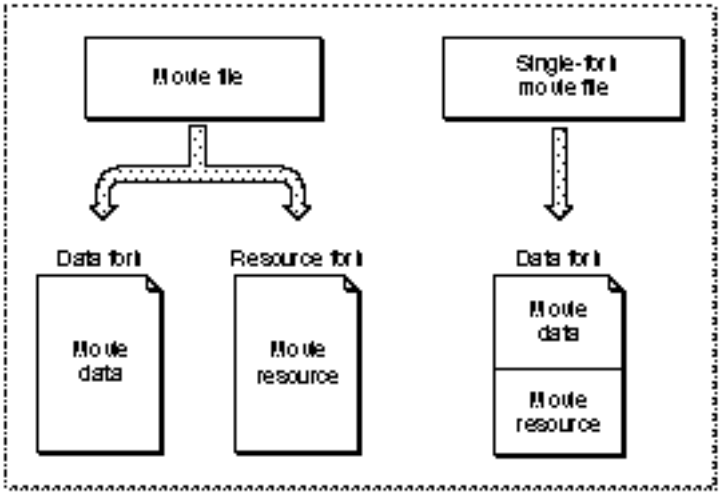
In the Macintosh file system, the Movie Toolbox typically uses both the resource fork and the data fork to store a QuickTime movie. The resource fork contains the movie resource. The data fork contains the actual movie data (or references to external data).

To facilitate data exchange between Macintosh computers and other computers, the Movie Toolbox can also understand movie files that store all the information for a movie in the data fork. These movie files are called **single-fork movie files.** Single-fork movie files are a possible way to export QuickTime movies to other systems, such as a computer using QuickTime for Windows.

Your application can create single-fork movie files from normal movie files by calling the Movie Toolbox's `FlattenMovieData` function (see the chapter "Movie Toolbox" for more information about this function). Your application can read single-fork movie files using the standard Movie Toolbox functions—you do not need to perform any special processing.

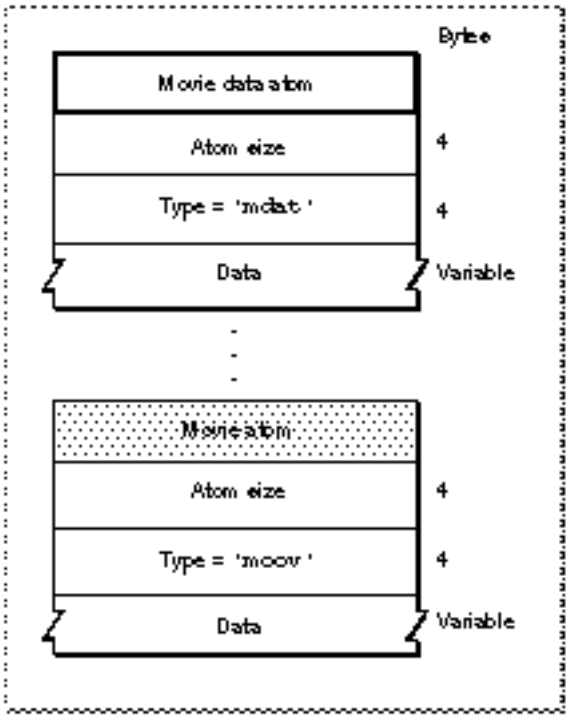
Figure 4-1 shows the difference between single-fork and normal movie files. A standard Macintosh movie file contains information in both the data and the resource forks. A single-fork movie file contains a data fork.

Figure 4-1 Movie files and single-fork movie files



Single-fork movie files store all the information for a movie in the data fork. The data fork contains two atoms: a movie data atom (`'mdat'`) and a movie resource atom. The movie's media data is stored in the movie data atom. Other atoms may follow the movie data atom. The movie resource atom follows the movie data atom and holds the description of the movie. There is no resource fork for this kind of movie file. Figure 4-2 shows the layout of a single-fork movie file. The movie data atom contains no other atoms, whereas the movie atom may contain other atoms.

Figure 4-2 The structure of a single-fork movie file



Atoms

The basic data unit in a QuickTime movie resource is the **atom.** Each atom contains size and type information along with its data. The `size` field indicates the number of bytes in the atom, including the `size` and `type` fields. The `type` field specifies the type of data stored in the atom and, by implication, the format of that data.

Atom Types

Table 4-1 lists the atom types defined by Apple and their corresponding constants and atom names.

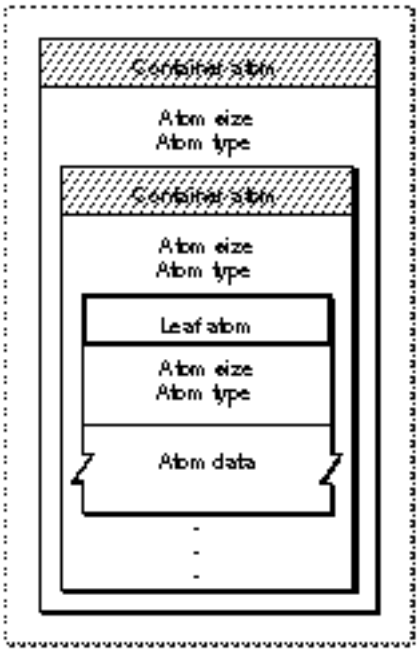
Table 4-1 Apple-defined atom types

Constant	Atom type	Atom name
MovieAID	'moov'	Movie atom
MovieHeaderAID	'mvhd'	Movie header atom
ClipAID	'clip'	Clipping atom
RgnClipAID	'crgn'	Clipping region atom
MatteAID	'matt'	Track matte atom
MatteCompAID	'kmat'	Compressed matte atom
TrackAID	'trak'	Track atom
UserDataAID	'udta'	User-defined data atom
TrackHeaderAID	'tkhd'	Track header atom
EditsAID	'edts'	Edit atom
EditsListAID	'elst'	Edit list atom
MediaAID	'mdia'	Media atom
MediaHeaderAID	'mdhd'	Media header atom
MediaInfoAID	'minf'	Media information atom
VideoMediaInfoHeaderAID	'vmhd'	Video media information header atom
SoundMediaInfoHeaderAID	'smhd'	Sound media information header atom
DataInfoAID	'dinf'	Data information atom
DataRefAID	'dref'	Data reference atom
SampleTableAID	'stbl'	Sample table atom
STSampleDescAID	'stsd'	Sample description atom
STTimeToSampAID	'stts'	Time-to-sample atom
STSyncSampleAID	'stss'	Sync sample atom
STShadowSyncAID	'stsh'	Shadow sync atom
STSampleToChunkAID	'stsc'	Sample-to-chunk atom
HandlerAID	'hdlr'	Handler reference atom
STSampleSizeAID	'stsz'	Sample size atom
STChunkOffsetAID	'stco'	Chunk offset atom

The Layout of a QuickTime Atom

Figure 4-3 shows the layout of a sample QuickTime atom. Each atom carries its own size and type information as well as its data. Throughout this chapter, the name of a **container atom** (an atom that contains other atoms, including other container atoms) is printed across a horizontal gray band, and the name of a **leaf atom** (an atom that contains no other atoms) is printed across a horizontal drop shadow box.
Leaf atoms contain data, usually in the form of tables.

Figure 4-3 A sample QuickTime atom

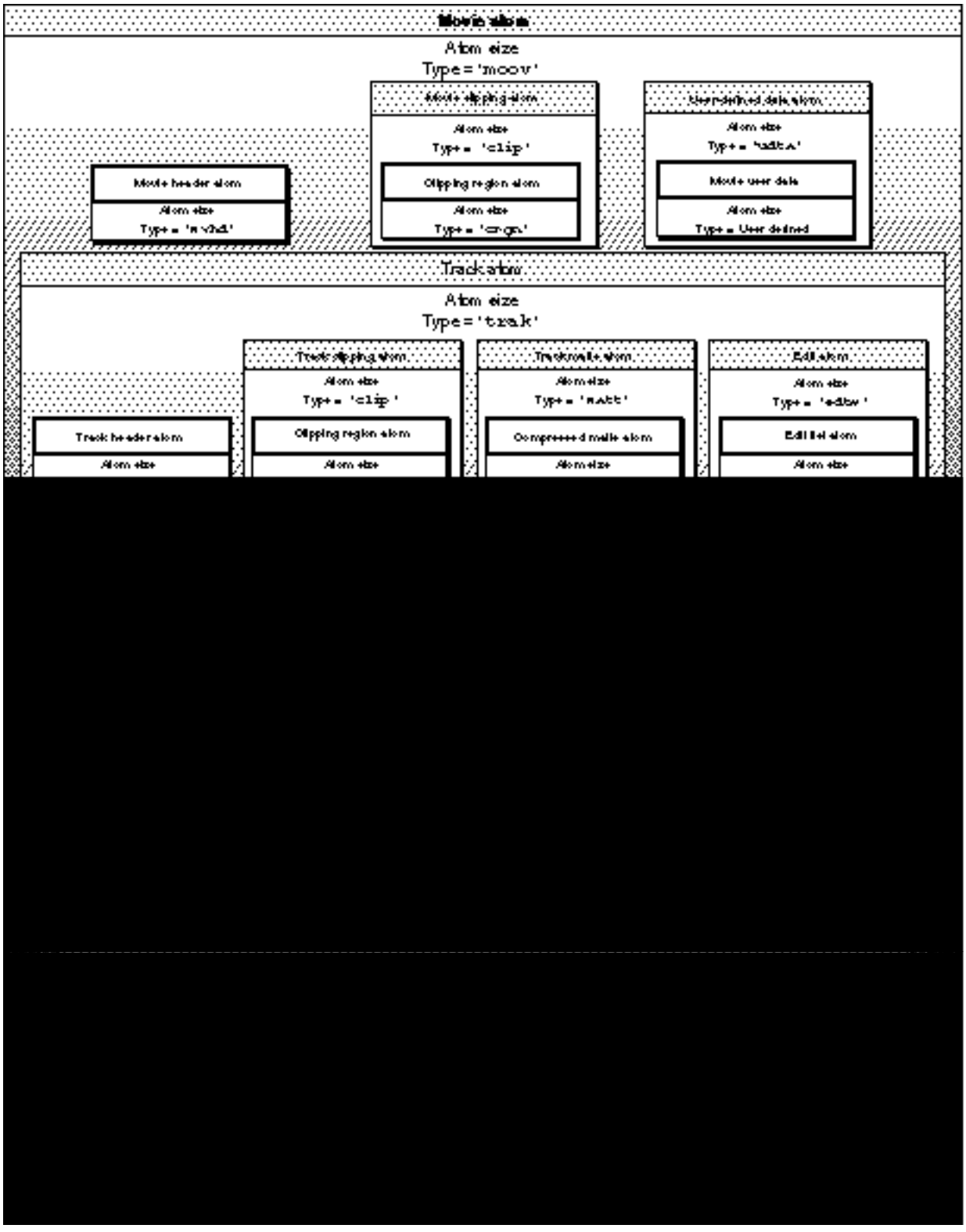


Overview of the Movie Resource Atom

Movie resources consist of movie atoms, which in turn contain track atoms, which in turn contain media atoms (see the chapter “Movie Toolbox” in this book for information about the relationships between movies, tracks, and media structures). Leaf atoms usually contain tables of data. For example, the track atom contains the edit atom, which contains a leaf atom called the *edit list atom*. The edit list atom contains an edit list table. (See Figure 4-15 on page 4-24 for an illustration of the edit atom, and see Figure 4-16 on page 4-25 for an illustration of the edit list table.)

Figure 4-4 provides a conceptual view of the organization of a QuickTime movie, which, in this case, has one track containing video information. Each nested box in the illustration represents an atom that belongs to the atom underlying it. The figure does not show the data regions of any of the atoms. These areas are described in the pertinent sections that follow.

Figure 4-4 Sample organization of a one-track video movie



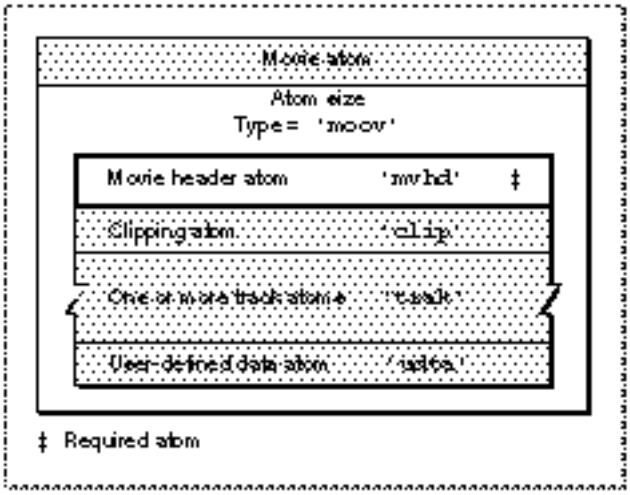
Movie Atoms

You can use movie atoms to specify information that defines a movie. The movie atom contains the movie header atom, which defines the time scale and duration information for the entire movie, as well as its display characteristics. In addition, the movie atom contains each track in the movie.

The movie atom has an atom type of `'moov'`. It contains other types of atoms, including one leaf atom—the movie header (`'mvhd'`)—and several atoms that contain other atoms: a clipping atom (`'clip'`), one or more track atoms (`'trak'`), and user-defined data (`'udta'`).

Figure 4-5 shows the layout of a movie atom. The movie header atom is the only required atom in the movie atom.

Figure 4-5 The layout of a movie atom



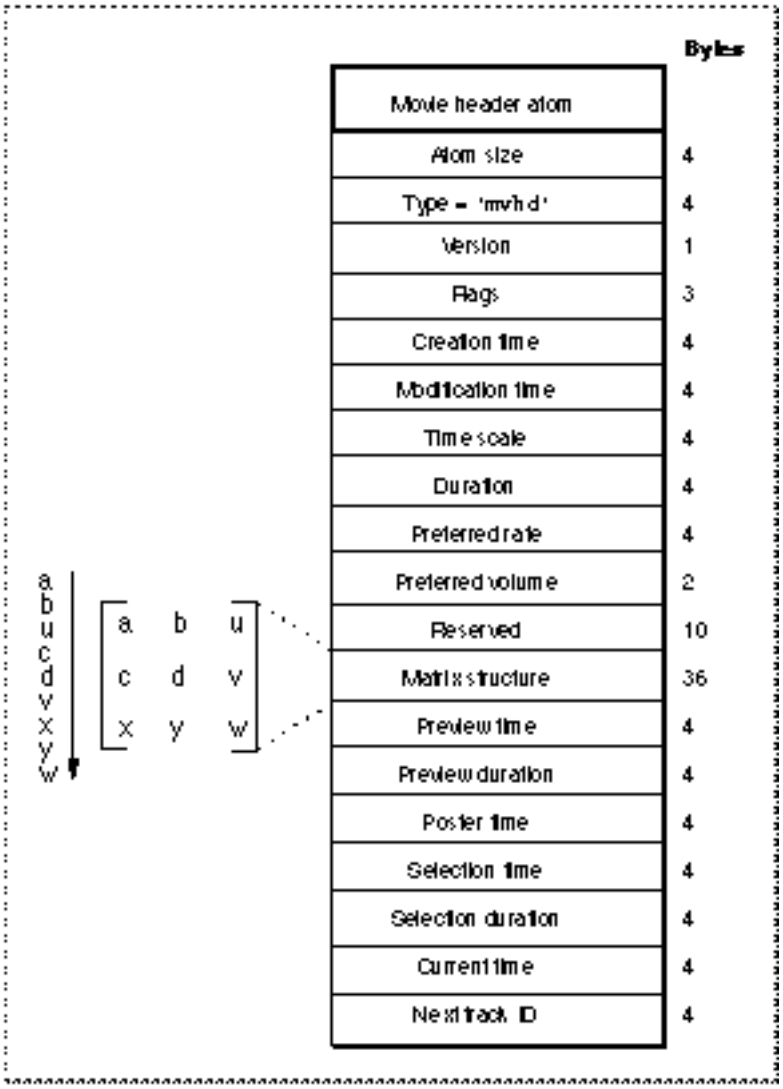
You define a movie atom by specifying these elements:

- Size. The number of bytes in this movie atom.
- Type. The type of this movie atom (the atom type, `'moov'`).
- Movie header. The movie header atom associated with this movie. See the next section for details on the movie header atom.
- Movie clipping atom. The clipping atom associated with this movie. See “Clipping Atoms,” which begins on page 4-22, for more information.
- Track list. One or more track atoms associated with this movie. See “Track Atoms,” which begins on page 4-13, for details on track atoms and their associated atoms.
- User data. The user-defined data atom associated with this movie. See “User-Defined Data Atoms,” which begins on page 4-19, for a discussion of user-defined data.

Movie Header Atoms

You can use the movie header atom to specify the characteristics of an entire movie. Figure 4-6 shows the layout of the movie header atom. The movie header atom is a leaf atom, which contains time information for the entire movie, such as time scale and duration. It also illustrates the data stream specified in the matrix structure field.

Figure 4-6 The layout of a movie header atom



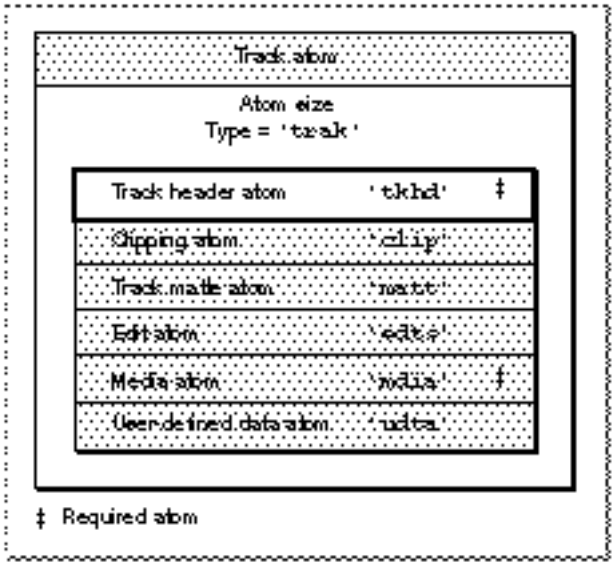
You define a movie header atom by specifying these elements:

- Size. A long integer that specifies the number of bytes in this movie header atom.
- Type. A long integer that specifies the format of the data in this movie header atom (defined by the atom type, `'mvhd'`).
- Version. A 1-byte specification of the version of this movie header atom.
- Flags. Three bytes of space for future movie header flags.
- Creation time. A long integer that specifies (in seconds since midnight, January 1, 1904) when the movie atom was created.
- Modification time. A long integer that specifies (in seconds since midnight, January 1, 1904) when the movie atom was changed.
- Time scale. A time value that indicates the **time scale** for this movie—that is, the number of time units that pass per second in its time coordinate system. A time coordinate system that measures time in sixtieths of a second, for example, has a time scale of 60.
- Duration. A time value that indicates the duration of the movie in time scale units.
- Preferred rate. A fixed number that specifies the rate at which to play this movie.
- Preferred volume. A 16-bit fixed number that specifies how loud to play this movie's sound.
- Reserved. Ten bytes reserved for use by Apple. Set to 0.
- Matrix. The matrix structure associated with this movie. A matrix shows how to map points from one coordinate space into another coordinate space. See the chapter "Movie Toolbox" in this book for details on matrix structures.
- Preview time. The time value in the movie at which the preview begins.
- Preview duration. The duration of the movie preview in movie time scale units. For more on time, see the chapter "Movie Toolbox" in this book.
- Poster time. The time value of the time of the movie poster.
- Selection time. The time value for the start time of the current selection.
- Selection duration. The duration of the current selection in movie time scale units.
- Current time. The time value for current time position within the movie.
- Next track ID. A long integer that indicates a value to use for the track ID number of the next track added to this movie.

Track Atoms

Track atoms define a single track of a movie. A movie may consist of one or more tracks. Each track is independent of the other tracks in the movie and carries its own temporal and spatial information. Each track atom contains its associated media atom. Figure 4-7 shows the layout of a track atom. The track atom requires the track header atom and the media atom.

Figure 4-7 The layout of a track atom



The track atom contains a track header atom (`'tkhd'`), a track clipping atom (`'clip'`), a track matte atom (`'matt'`), an edit atom (`'edts'`), a media atom (`'mdia'`), and a user-defined data atom (`'udta'`). You define a track atom by specifying these elements:

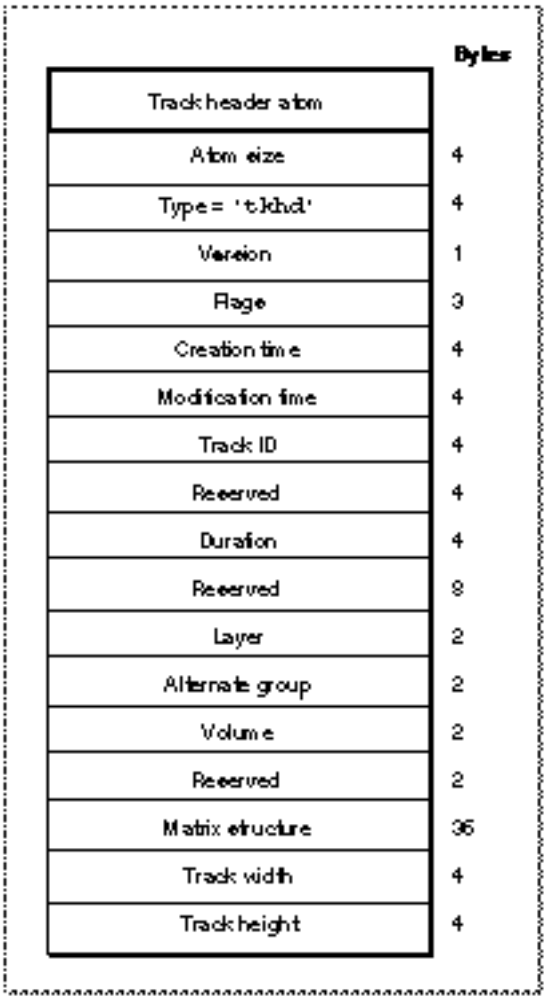
- Size. The number of bytes in this track atom.
- Type. The type of this track atom (the atom type, `'trak'`).
- Track header. The track header atom associated with this track. See the next section for details.
- Track clipping. The track clipping atom associated with this track. See “Clipping Atoms,” which begins on page 4-22, for more on clipping atoms.
- Track matte. The track matte atom associated with this track. See “Track Matte Atoms,” which begins on page 4-23, for more on track matte atoms.
- Edits. The edit atom associated with this track. See “Edit Atoms,” which begins on page 4-24, for details.

- Media. The media atom associated with this track. See “Media Atoms,” which begins on page 4-16, for details.
- User data. The user-defined data atom associated with this track. This field is used for extension with new data types. See “User-Defined Data Atoms,” which begins on page 4-19, for details.

Track Header Atoms

The track header atom specifies the characteristics of a single track within a movie. A track header atom contains a `size` field that specifies the number of bytes and a `type` field that indicates the format of the data (defined by the atom type, `'tkhd'`). Figure 4-8 shows the structure of the track header atom.

Figure 4-8 The layout of a track header atom



The track header atom contains the track characteristics for the track, including temporal, spatial, and volume information. You define a track header atom by specifying these elements:

- Size. A long integer that specifies the number of bytes in this track header atom.
- Type. A long integer that specifies the type of data in this track header atom (defined by the atom type, `'tkhd'`).
- Version. A 1-byte specification of the version of this track header.
- Track header flags. Three bytes that are reserved for the track header flags, which adjust the remaining fields in the track header according to the kind of movie track you specify with the following enumeration:

```
enum
{
   TrackEnable = 1<<0,  /* enabled track */
   TrackInMovie = 1<<1, /* track in playback */
   TrackInPreview = 1<<2,/* track in preview */
   TrackInPoster = 1<<3 /* track in poster */
};
```

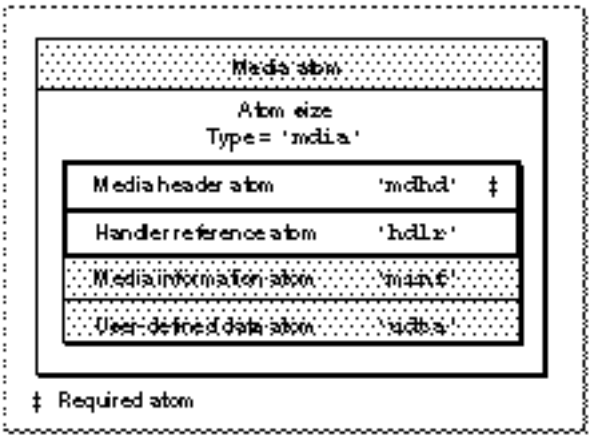
- Creation time. A long integer that indicates (in seconds since midnight, January 1, 1904) when the track header was created.
- Modification time. A long integer that indicates (in seconds since midnight, January 1, 1904) when the track header was changed.
- Track ID. A long integer that specifies the value to use for the track ID number.
- Reserved. A long integer that is reserved for use by Apple. Set the value of this field to 0.
- Duration. The duration of this track (in movie time).
- Reserved. An 8-byte value that is reserved for use by Apple. Set the value of this field to 0.
- Layer. The priority of playing this track in a movie. When it plays a movie, the Movie Toolbox displays the movie's tracks according to their **layer**—tracks with lower layer numbers are displayed in front; tracks with higher layer numbers are displayed in back.
- Alternate group. A short integer that specifies a collection of movie tracks that contain alternate data for one another. QuickTime chooses one track from the group to be used when the movie is played. The choice may be based on such considerations as playback quality or language and the capabilities of the computer.
- Volume. A short integer that indicates how loudly this track's sound is to be played.
- Reserved. A short integer that is reserved for use by Apple. Set the value of this field to 0.
- Matrix. The matrix structure associated with this track. See Figure 4-6 on page 4-11 for an illustration of a matrix structure.

- Track width. A fixed number that specifies the width of this track.
- Track height. A fixed number that indicates the height of this track.

Media Atoms

Media atoms define the data for a movie track. The media atom contains information that specifies the component that is to interpret the media data, and it also specifies the data references. Figure 4-9 shows the layout of a media atom.

Figure 4-9 The layout of a media atom



The media atom has an atom type of `'mdia'`. It may contain other atoms, such as a media header (`'mdhd'`), a handler reference (`'hdlr'`), media information (`'minf'`), and user-defined data (`'udta'`). The only required atom in a media atom is the media header atom.

Note
The handler reference atom lets you know what kind of media this media atom contains—for example, video or sound. The layout of the media information atom is specific to the media handler that is to interpret the media. "Media Information Atoms," which begins on page 4-26, discusses how data may be stored in a media, using the video media format defined by Apple as an example. u

You define a media atom by specifying these elements:

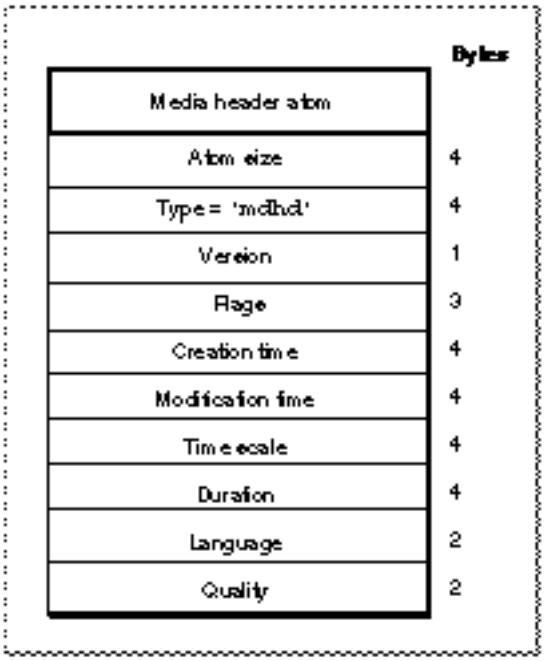
- Size. A long integer that specifies the number of bytes in this media atom.
- Type. A long integer that specifies the type of the data in this media atom (defined by the `'mdia'` atom type).
- Media header. The media header atom, which is described in the next section. It contains the standard media information.

- Media handler. The media handler, which is defined by the handler reference atom, described in “Handler Reference Atoms,” which begins on page 4-18.
- Media information. The media information atom. For an example of a media information atom, see “Video Media Information Atoms,” which begins on page 4-26.
- User data. The user-defined data atom associated with this media. This field is used for extension with new data types. See “User-Defined Data Atoms,” which begins on page 4-19, for details.

Media Header Atoms

The media header atom specifies the characteristics of the media that is used to store data for the movie track defined in its associated track atom. The media header atom contains the number of bytes in the media header atom, the format of the data in the media header atom (defined by the `'mdhd'` atom type), and the media header. The media characteristics include temporal information. Figure 4-10 shows the layout of the media header atom.

Figure 4-10 The layout of a media header atom



You define a media header atom by specifying these elements:

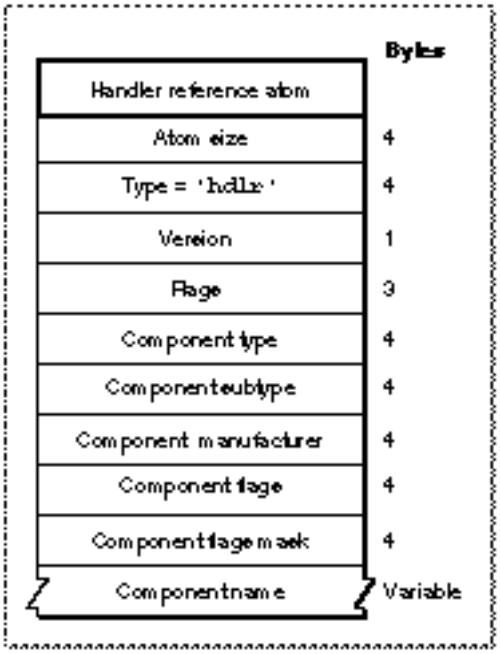
- Size. A long integer that specifies the number of bytes in this media header atom.
- Type. A long integer that specifies the type of data in this media header atom (defined by the atom type, `'mdhd'`).

- Version. One byte that specifies the version of this movie.
- Flags. Three bytes of space for future media header flags.
- Creation time. A long integer that specifies (in seconds since midnight, January 1, 1904) when the media atom was created.
- Modification time. A long integer that specifies (in seconds since midnight, January 1, 1904) when the media atom was changed.
- Time scale. A time value that indicates the time scale for this media—that is, the number of time units that pass per second in its time coordinate system.
- Duration. The duration of this media in media time scale units.
- Language. A short integer that specifies the language code for this media. See *Inside Macintosh: Text* for more on language codes.
- Quality. A short integer that specifies the playback quality (that is, its suitability for playback in a given environment) for this media. See the chapter "Movie Toolbox" in this book for details on playback quality.

Handler Reference Atoms

The handler reference atom specifies the component that is to interpret a media's data. This component is called a *media handler*. (See the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for more information about components.) Figure 4-11 shows the layout of a handler reference atom.

Figure 4-11 The layout of a handler reference atom



You define a handler reference atom by specifying these elements:

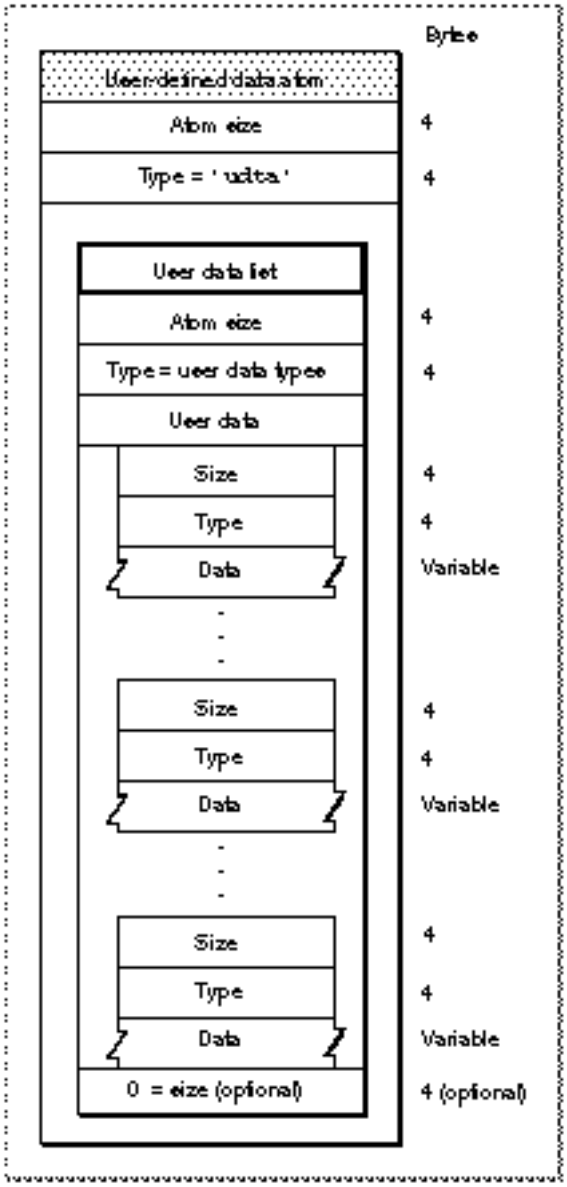
- Size. The number of bytes in this handler reference atom.
- Type. The type of the data (defined by the `'hdlr'` atom type) in the handler reference atom.
- Flags. A 1-byte specification of the version of this handler information.
- Version. A 3-byte space for future handler information flags.
- Component type. A four-character code that identifies the type of the media handler. All components of a particular type must support a common set of functions. Examples of component types are `'mhlr'` and `'dhlr'`.
- Component subtype. A four-character code that identifies the type of the media handler. Different types of a component type may support additional features or provide interfaces that extend beyond the standard functions for a given component type value. For media handlers, this field defines the type of data—for example, `'vide'` or `'soun'`.
- Component manufacturer. A four-character code that identifies the manufacturer of this media handler. This field allows for further differentiation between individual components. For example, components made by a specific manufacturer may support an extended feature set.
- Component flags. A 32-bit field that provides additional information about a particular media handler. The most significant 8 bits are reserved for use by the Component Manager and provide both static and dynamic information about the component.
- Component flags mask. A 32-bit field that indicates which flags in the component mask are relevant to a particular search operation. These flags are used when searching for a handler component.
- Component name. A Pascal string that specifies the name of the component—that is, the original handler used when this movie was created.

User-Defined Data Atoms

Many movie, track, and media atoms contain atoms that store user-defined data. Your application may store data in these user-defined data atoms.

Figure 4-12 shows the layout of a user-defined data atom.

Figure 4-12 The layout of a user-defined data atom



You define a user-defined data atom by specifying these elements:

- Size. The number of bytes in the data element.
- Type. The type of the data in the data element (defined by the `'udta'` atom type).
- User data list. The movie user data atom contains a user data list that is itself formatted like a series of atoms. Each data element in the private data portion of the user-defined data atom contains size and type information along with the data. Furthermore, the list of atoms is optionally terminated by a 0.

The following user data types are currently defined:

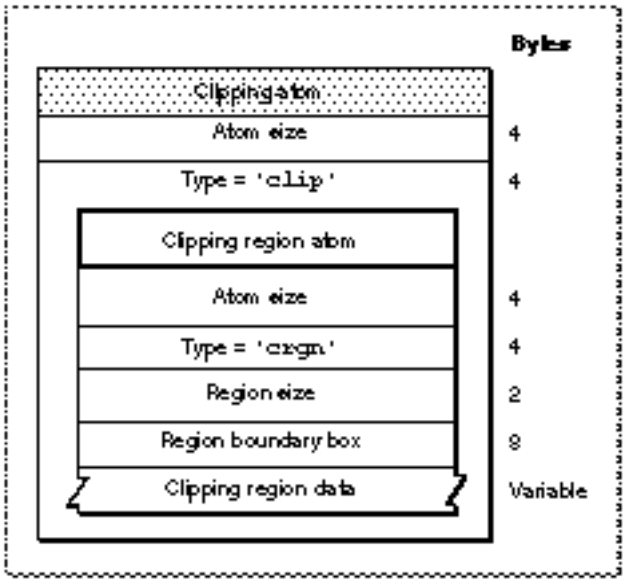
`'©cpy'`	Copyright statement
`'©day'`	Date the movie content was created
`'©dir'`	Name of movie's director
`'©ed1' to '©ed9'`	Edit dates and descriptions
`'©fmt'`	Indication of movie format (computer-generated, digitized, and so on)
`'©inf'`	Information about the movie
`'©prd'`	Name of movie's producer
`'©prf'`	Names of performers
`'©req'`	Special hardware and software requirements
`'©src'`	Credits for those who provided movie source content
`'©wrt'`	Name of movie's writer

User data items of these types must contain text data only.

Clipping Atoms

Clipping atoms specify the clipping regions for movies and for tracks. Figure 4-13 shows the layout of clipping atoms.

Figure 4-13 The layout of a clipping atom



You define a clipping atom by specifying these elements:

- Size. The number of bytes in this clipping atom.
- Type. The type of the data in this clipping atom (defined by the `'clip'` atom type).
- Clipping region atom. Described in the next section.

Clipping Region Atoms

The clipping region atom specifies the clipping data. The layout of the clipping region atom is shown in Figure 4-13. You define a clipping region atom by specifying these elements:

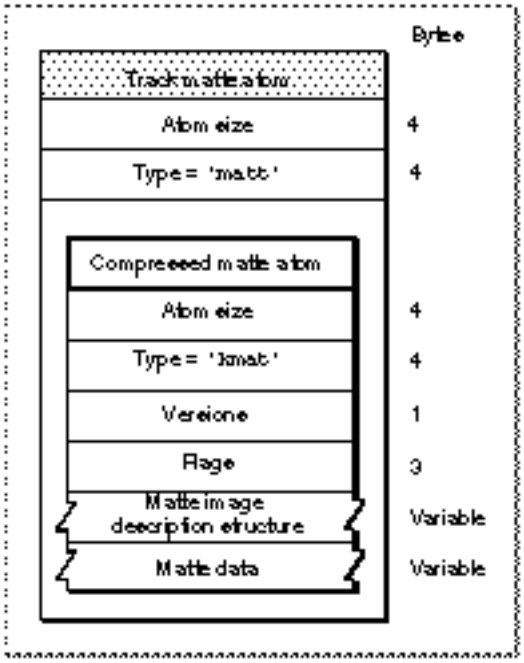
- Size. A long integer that indicates the number of bytes in the clipping region data atom.
- Type. A long integer that indicates the type of the clipping region data (defined by the `'crgn'` atom type).

The region size, region boundary box, and data fields constitute a QuickDraw region. See the chapter "QuickDraw" in *Inside Macintosh: Imaging* for details on QuickDraw regions.

Track Matte Atoms

Track matte atoms specify the mattes for tracks. (A **track matte** is a pixel map that defines the blending of visual track data. See the chapter “Movie Toolbox” in this book for details.) Figure 4-14 shows the layout of track matte atoms.

Figure 4-14 The layout of a track matte atom



You define a track matte atom by specifying these elements:

- Size. A long integer that specifies that number of bytes in this track matte atom.
- Type. A long integer that specifies the type of this track matte atom (defined by the `'matt'` atom type).
- Compressed matte atom. The compressed matte atom, which is described in the next section.

Compressed Matte Atoms

The compressed matte atom specifies the image description structure associated with a particular matte atom. The layout of the compressed matte atom is shown in Figure 4-14.

You define a compressed matte atom by specifying these elements:

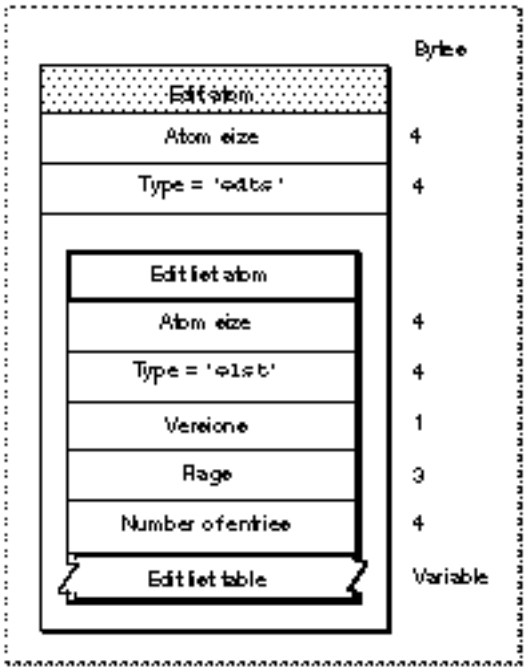
- Size. A long integer that indicates the number of bytes in this compressed matte atom.
- Type. A long integer that indicates the type of the data in this atom (defined by the `'kmat'` atom type).
- Version. A 1-byte specification of the version of this compressed matte atom.
- Flags. Three bytes of space for future flags associated with this compressed matte atom.
- Matte image description. An image description structure of variable length and associated with this matte data. See the chapter “Image Compression Manager” in this book for details on the image description structure.
- Matte data. The compressed matte data, which is of variable length.

Edit Atoms

You can use edit atoms to define the portions of the media that are to be used to build up a track for a movie. Figure 4-15 shows the layout of an edit atom.

Note
If the edit atom or the edit list atom is missing, you can assume that the entire media is contained in the track. u

Figure 4-15 The layout of an edit atom



You define an edit atom by specifying these elements:

- Size. A long integer that indicates the number of bytes in this edit atom.
- Type. A long integer that indicates the type of the data in this edit atom (defined by the `'edts'` atom type).
- Edit list. The edit list atom that contains the edit list information, described in the next section.

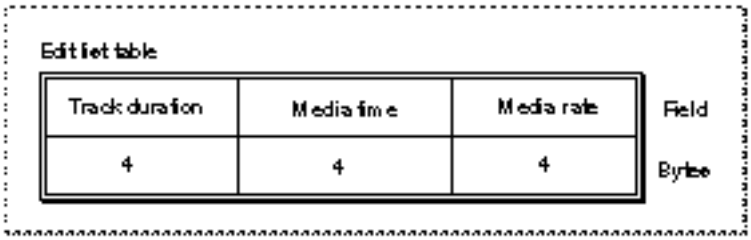
Edit List Atoms

You can use the edit list atom, also shown in Figure 4-15, to tell QuickTime how to map from a time in a movie to a time in a media, and ultimately to media data. This information is in the form of an edit list table, shown in Figure 4-16.

You define an edit list atom by specifying the following elements:

- Size. A long integer that specifies the number of bytes in the edit list atom.
- Type. A long integer that specifies the type of the edit list data (defined by the `'elst'` atom type).
- Version. A 1-byte specification of the version of this edit list atom.
- Flags. Three bytes of space for future flags to be associated with this edit list atom.
- Number of entries. The number of entries in the edit list atom.
- Edit list table. Each entry in the edit list table (shown in Figure 4-16) describes a single edit and contains a track duration field, a media time field, and a media rate field.

Figure 4-16 The layout of an edit list table



You create an edit list table by specifying these elements:

- Track duration. The duration of this edit segment in movie time scale units.
- Media time. The starting time within the media of this edit segment (in media time scale units). If –1, it is an empty edit.
- Media rate. A fixed number that specifies the relative rate at which to play the media for this edit segment.

Media Information Atoms

Media information atoms (defined by the `'minf'` data type) store handler-specific information for the media data that constitutes a track. The media handler uses this information to map from media time to media data. These atoms are formatted differently based on the type of media data stored in the atom. The format and content of media information atoms are dictated by the media handler that is responsible for interpreting the media data stream. Another media handler would not know how to interpret this information. This section describes examples of atoms that store media information for the video (defined by the `'vmhd'` atom type) and sound (defined by the `'smhd'` atom type) portions of QuickTime movies.

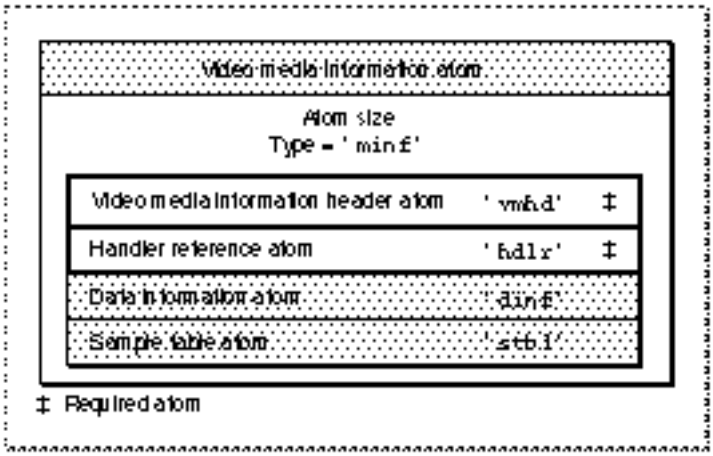
Note

"Using Media Information Atoms," which begins on page 4-45, discusses how the video media handler locates samples in a video media. u

Video Media Information Atoms

Video media information atoms are the highest-level atoms in video media. A number of other atoms define specific characteristics of the video media data. Figure 4-17 shows the layout of a video media information atom.

Figure 4-17 The layout of a media information atom for video



You define a video media information atom by specifying these elements:

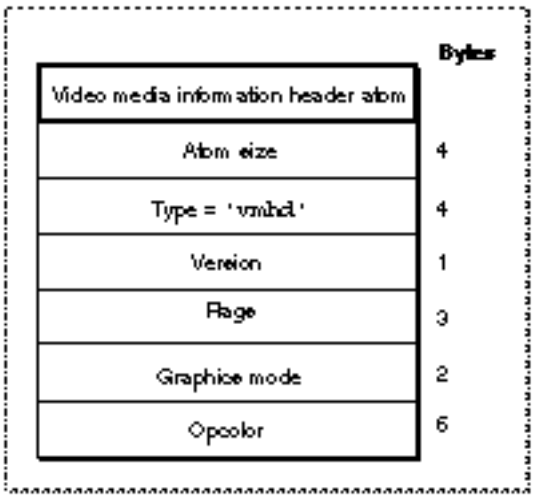
- Size. A long integer that specifies the number of bytes in this video media information atom.

- Type. A long integer that specifies the type of the data (defined by the `'minf'` atom type) in this media information header.
- Video media information. The video media information header atom (a required atom), which is described in the next section.
- Handler reference. The handler reference atom (a required atom), which contains information specifying the data handler component that provides access to the media data. See the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for more information about components. Figure 4-11 on page 4-18 shows the layout of a handler reference atom. The handler reference uses the data information atom, described by the `datainfo` field in the video media information structure.
- Data information. The data information atom, described in "Data Information Atoms" on page 4-30.
- Sample table. The sample table atom, described in "Sample Table Atoms" on page 4-33.

Video Media Information Header Atoms

Video media information atoms are the highest-level atoms in video media. A number of other atoms define specific characteristics of the video media data. Figure 4-18 shows the structure of a video media information header atom.

Figure 4-18 The layout of a media information header atom for video



You define a video media information header atom by specifying these elements:

- Size. A long integer that specifies the number of bytes in the media information in this video media information header.

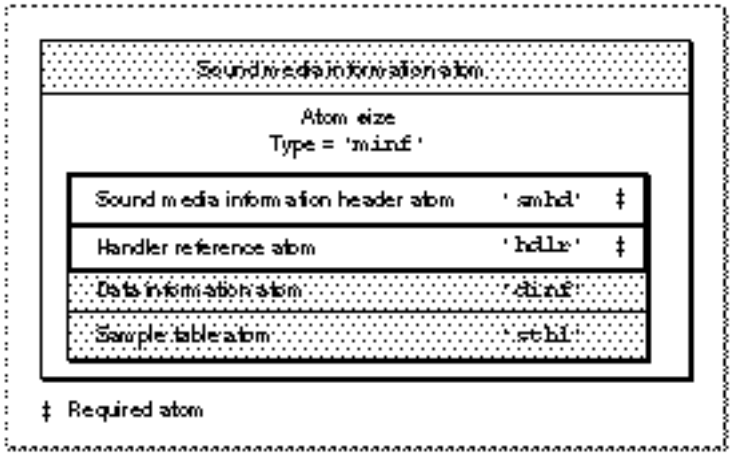
- Type. A long integer that specifies the type of the data (defined by the `'vmhd'` atom type) in this video media information header.
- Version. A 1-byte specification of the version of this video media information header.
- Flags. A 3-byte space for video media information flags. The `videoFlagNoLeanAhead` flag is available, which instructs QuickTime that the video was not created skewed and that it should use a technique having greater accuracy.
- Graphics mode. A short integer that specifies the transfer mode, which is a specification of which Boolean operation QuickDraw should perform when drawing or transferring an image from one location to another.
- Opcolor. Three 16-bit values that specify the red, green, and blue colors for the transfer mode operation indicated in the graphics mode field.

For comprehensive details on QuickDraw's transfer modes and opcolors and their values, see *Inside Macintosh: Imaging*.

Sound Media Information Atoms

Sound media information atoms are the highest-level atoms in sound media. These atoms define specific characteristics of the sound media data. Figure 4-19 shows the layout of a sound media information atom.

Figure 4-19 The layout of a media information atom for sound



In addition to the size and type information, the sound media information atom contains the sound media information header atom, which is described in the next section, and the handler reference atom, the data information atom, and the sample table atom.

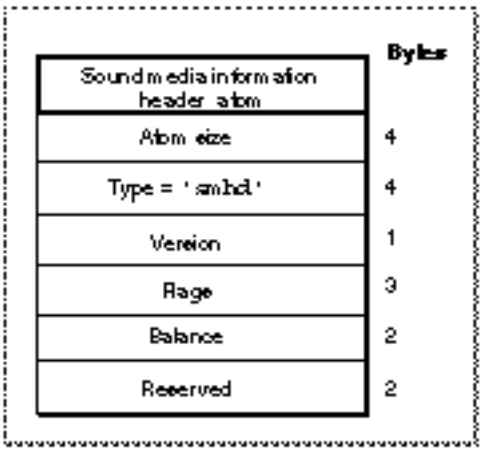
You define a sound media information atom by specifying these elements:

- Size. A long integer that specifies the number of bytes in this sound media information atom.
- Type. A long integer that specifies the type of the data in this sound media information header (defined by the `'minf'` data type).
- Sound media information. The sound media information header atom (a required atom), which is described in the next section.
- Handler reference. The handler reference atom (a required atom), which contains information specifying the data handler component that provides access to the media data. See the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for more information about components. Figure 4-11 on page 4-18 shows the layout of a handler reference atom. The handler reference atom uses the data information atom, described by the `dataInfo` field in this sound media information structure.
- Data information. The data information atom, described in "Data Information Atoms," which begins on page 4-30.
- Sample table. The sample table atom, described in "Sample Table Atoms," which begins on page 4-33.

Sound Media Information Header Atoms

The sound media information header atom (shown in Figure 4-20) stores the sound media information.

Figure 4-20 The layout of a sound media information header atom



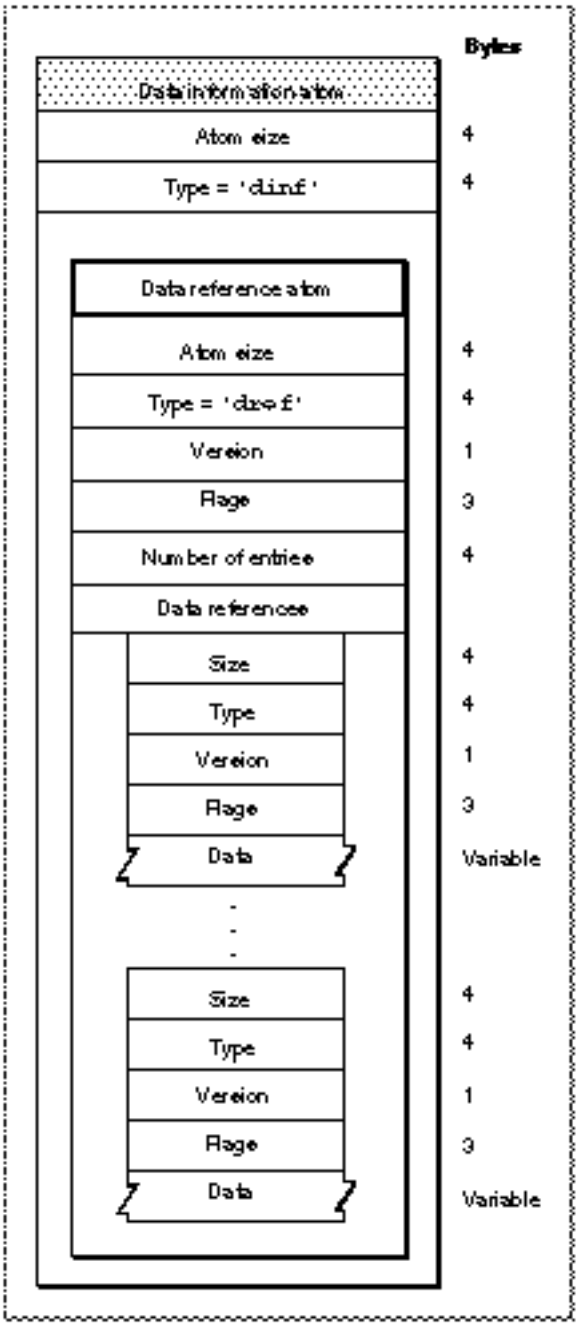
You define a sound media information header atom by specifying these elements:

- Size. A long integer that specifies the number of bytes in this sound media information header atom.
- Type. A long integer that specifies the type of the data in this sound media information header atom (defined by the `'smhd'` data type).
- Version. A 1-byte specification of the version of this sound media information header.
- Flags. Three bytes of space for future associated flags.
- Balance. A short integer that specifies the sound balance of this sound media. (Sound balance is the setting that controls the mix of sound between the two speakers of a computer.) This field is normally set to 0. See the chapter "Movie Toolbox" in this book for more on sound balance.
- Reserved. Reserved for use by Apple. Set this field to 0.

Data Information Atoms

The handler reference atom (described in "Handler Reference Atoms," which begins on page 4-18) contains information specifying the data handler component that provides access to the media data. See the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for more about components. The handler uses the data information atom, which you can use to specify where the media data is stored. Figure 4-21 shows the layout of the data information atom.

Figure 4-21 The layout of a data information atom



You define a data information atom by specifying these elements:

- Size. A long integer that specifies the number of bytes in this data information atom.
- Type. A long integer that specifies the format (defined by the `'dinf'` atom type) of the data in this data information atom.

- Data references. The data reference atom, described in the next section, contains the data references.

Data Reference Atoms

Figure 4-21 also shows the data reference atom, which encompasses the data references.

You define a data reference atom by specifying these elements:

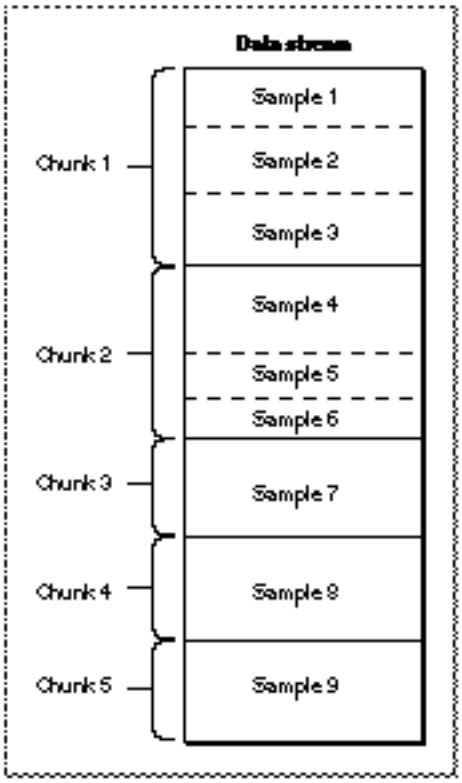
- Size. A long integer that specifies the number of bytes in this data reference container atom.
- Type. A long integer that specifies the type of the data in the data reference atom (defined by the `'dref'` data type).
- Version. A 1-byte specification of the version of this data reference atom.
- Flags. Three bytes that contain space for future flags.
- Number of entries. A count of entries in the data references field.
- Data references. Data references are formatted like atoms, as follows:
  - Size. A long integer that specifies the number of bytes in these data references.
  - Type. A long integer that specifies the type of the data (currently defined by the `'alis'` data type on the Macintosh computer) in the data references.
  - Version. A 1-byte specification of the version of these data references.
  - Flags. Three bytes that contain the attributes of the data in these data references. One enumerated constant is available. The `dataRefSelfReference` attribute denotes that the data comes from the same location as the movie resource. If the movie resource came from a resource fork, the movie data is in the data fork of the same file. In the case of a single-fork file, the movie data is also in the data fork of the file.
  - Data references. The data reference information. (For the current data handlers, this is an alias).

An Introduction to Samples

One way to describe a sample (that is, a single element of a sequence of time-ordered data) is to include it in a sample table atom. Samples are stored sequentially in the media, and they may have varying durations. This approach enforces an ordering of the samples—it does not mean the sample data must be stored sequentially with respect to movie time in the actual data stream. Figure 4-22 shows the way that samples are stored in a series of **chunks** in a media. Chunks are a collection of data samples in a media that

allow optimized data access. A chunk may contain one or more samples. Chunks in a media may have different sizes, and the samples within a chunk may have different sizes.

Figure 4-22 Samples in a media



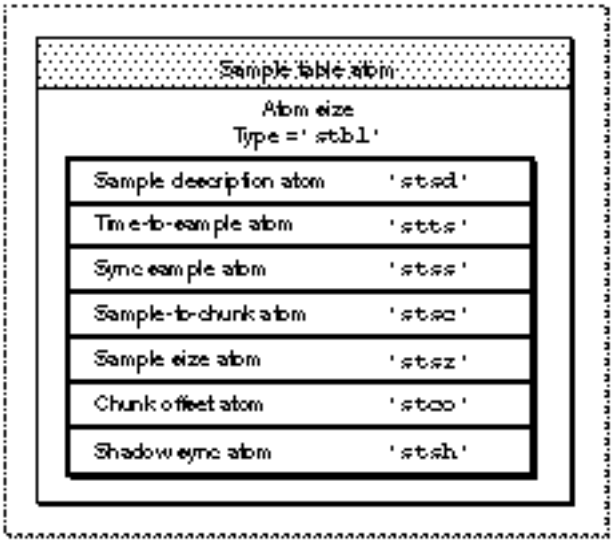
Sample Table Atoms

The sample table atom contains information for converting from media time to sample number to sample location. This atom also indicates how to interpret the sample (for example, whether to decompress the video sample and, if so, how). This section describes the format and content of the sample table atom.

The sample table has an atom type of `'stbl'`. It contains the sample description atom, the time-to-sample atom, the sample-to-chunk atom, the sync sample atom, the sample size atom, the chunk offset atom, and the shadow sync atom.

Figure 4-23 shows the layout of the sample table atom.

Figure 4-23 The layout of a sample table atom



You define a sample table atom by specifying these elements:

- Size. A long integer that specifies the number of bytes in the sample table atom.
- Type. A long integer that specifies the type of the data (defined by the `'stbl'` atom type) in the sample table atom.
- Sample description. The sample description atom, described in the next section.
- Time-to-sample. The time-to-sample atom, described in “Time-to-Sample Atoms,” which begins on page 4-36.
- Sync sample. The sync sample atom, described in “Sync Sample Atoms,” which begins on page 4-38.
- Sample-to-chunk. The sample-to-chunk atom, described in “Sample-to-Chunk Atoms,” which begins on page 4-39.
- Sample size. The sample size atom, described in “Sample Size Atoms,” which begins on page 4-41.
- Chunk offset. A chunk offset atom, described in “Chunk Offset Atoms,” which begins on page 4-42.
- Shadow sync. The shadow sync atom, described in “Shadow Sync Atoms,” which begins on page 4-44.

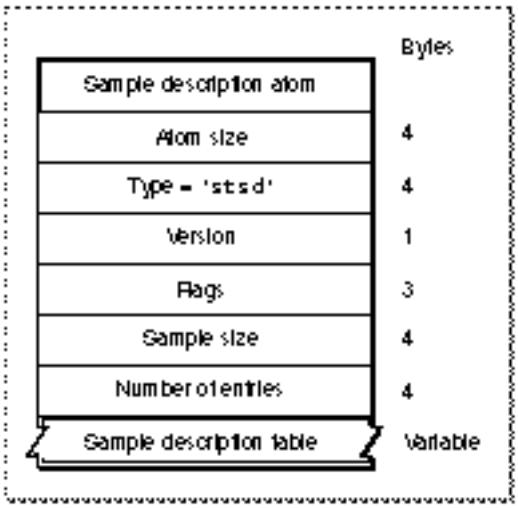
The following sections discuss each of the atoms that may be contained in a sample table.

Sample Description Atoms

The sample description atom stores information for the decoding of samples in the media. In the case of video media, the sample descriptions are image description structures (see the chapter “Image Compression Manager” earlier in this book for more information about image descriptions). Figure 4-24 shows the layout of the sample description atom.

The sample description atom has an atom type of `'stsd'`. The sample description atom contains a table of sample descriptions, each of which contains a single sample description. A media may have one or more sample descriptions, depending upon the number of different compression types used in the media. The sample-to-chunk atom identifies the sample description for each sample in the media by specifying the index into this table for the appropriate description (see “Sample-to-Chunk Atoms,” which begins on page 4-39).

Figure 4-24 The layout of a sample description atom



You define a sample description atom by specifying these elements:

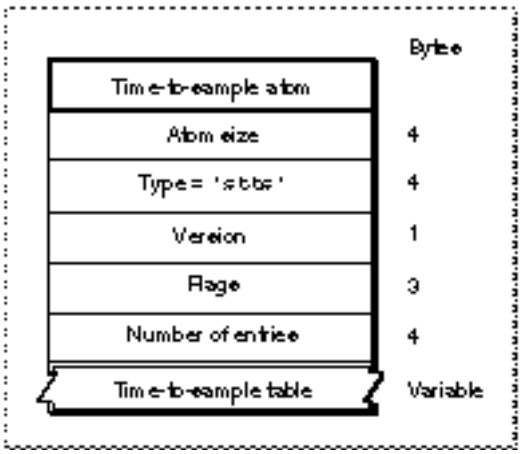
- Size. A long integer that specifies the number of bytes in this sample description atom.
- Type. A long integer that specifies the type (defined by the atom type `'stsd'`) of the data in this sample description atom.
- Version. A 1-byte specification of the version number of this sample description atom.
- Flags. Three bytes of space for future flags associated with it.
- Number of entries. A long integer that specifies how many entries in the sample description table are listed in the sample description table field of this atom.
- Sample description table. The sample description table, which contains a list of sample descriptions.

Time-to-Sample Atoms

Time-to-sample atoms store duration information for the samples in a media, providing a mapping from a time in a media to the corresponding data sample. The time-to-sample atom has an atom type of `'stts'`.

You can determine the appropriate sample for any given time in a media by examining the time-to-sample atom (shown in Figure 4-25), which contains the time-to-sample atom table.

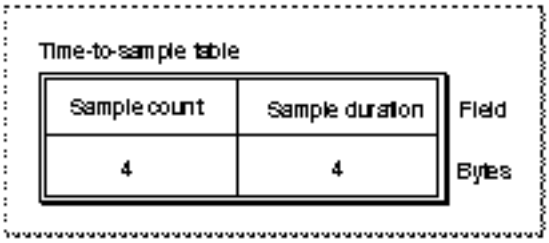
Figure 4-25 The layout of a time-to-sample atom



You define a time-to-sample atom by specifying these elements:

- Size. A long integer that specifies the number of bytes in this time-to-sample atom.
- Type. A long integer that specifies the type (defined by the `'stts'` atom type) of the data contained in the time-to-sample atom.
- Version. A 1-byte specification of the version number of this time-to-sample atom.
- Flags. Three bytes of space for any future flags associated with this time-to-sample atom.
- Number of entries. A long integer that specifies the number of entries in the time-to-sample table.
- Time-to-sample table. The time-to-sample atom contains a table that defines the duration of each sample in the media. Each table entry contains a count field and a duration field. The structure of the time-to-sample table is shown in Figure 4-26.

Figure 4-26 The layout of a time-to-sample table



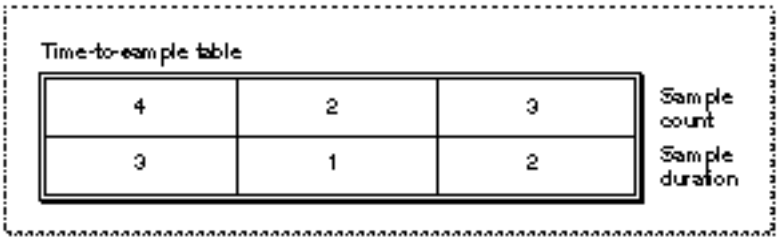
You define a time-to-sample table by specifying these entries:

- Sample count. A long integer that specifies the number of consecutive samples that have the same duration.
- Sample duration. A long integer that specifies the duration of each sample.

Entries in the table collect samples according to their order in the media and their duration. If consecutive samples have the same duration, a single table entry may be used to define more than one sample. In these cases, the count field indicates the number of consecutive samples that have the same duration. For example, if a video media has a constant frame rate, this table would have one entry.

Figure 4-27 shows an example of a time-to-sample table that is based on the data stream shown in Figure 4-22 on page 4-33. Figure 4-22 shows a total of nine samples that correspond in count and duration to the entries of the table shown in Figure 4-27. Even though samples 4, 5, and 6 are in the same chunk, sample 4 has a duration of 3, and samples 5 and 6 have a duration of 2.

Figure 4-27 An example of a time-to-sample table



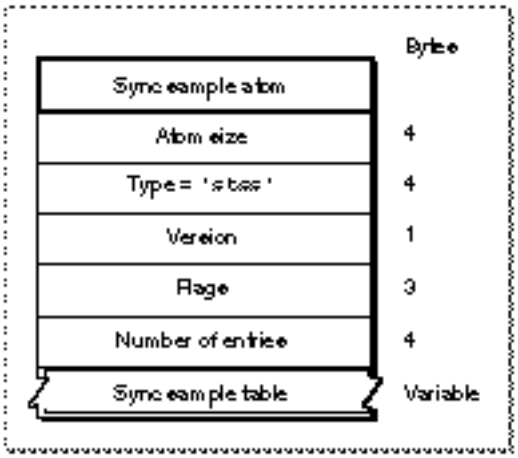
Sync Sample Atoms

The sync sample atom identifies the **key frames** in the media. In a media that contains compressed data, key frames define starting points for portions of a temporally compressed sequence (see the chapter “Image Compression Manager” in this book for more information about key frames and temporal compression in video data). The key frame is self-contained—that is, it is independent of preceding frames. Subsequent frames may depend on the key frame.

Sync sample atoms have an atom type of `'stss'`. The sync sample atom contains a table of sample numbers. Each entry in the table identifies a sample that is a key frame for the media. Figure 4-28 shows the layout of a sync sample atom.

If no sync sample atom exists, then all the samples are key frames.

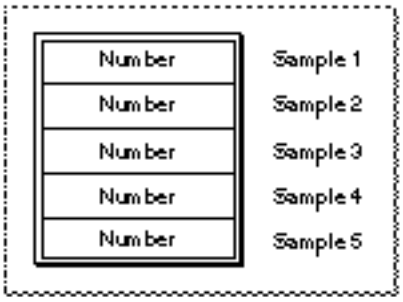
Figure 4-28 The layout of a sync sample atom



You define a sync sample atom by specifying these elements:

- Size. A long integer that specifies the number of bytes in this sync sample atom.
- Type. A long integer that specifies the type of the data of this sync sample atom (defined by the `'stss'` atom type).
- Version. A 1-byte specification of the version of this sync sample atom.
- Flags. Three bytes of space for future flags.
- Number of entries. A long integer that specifies how many sample numbers are in the sync sample table contained in the sync sample table field.
- Sync sample table. The sync sample table (shown in Figure 4-29) consists of an array of sample numbers. Each entry in the table identifies a sample that is a key frame for the media.

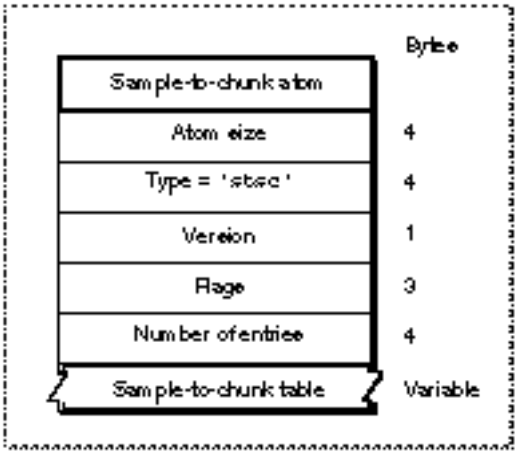
Figure 4-29 The layout of a sync sample table



Sample-to-Chunk Atoms

As samples are added to a media, they are collected into chunks that allow optimized data access. A chunk may contain one or more samples. Chunks in a media may have different sizes, and the samples within a chunk may have different sizes. The sample-to-chunk atom stores chunk information for the samples in a media. Figure 4-30 shows the layout of the sample-to-chunk atom. By examining the sample-to-chunk atom, you can determine the chunk that contains a specific sample.

Figure 4-30 The layout of a sample-to-chunk atom

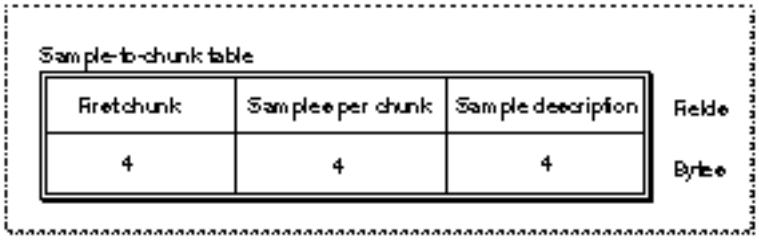


You define a sample-to-chunk atom by specifying these elements:

- Size. A long integer that specifies the number of bytes in this sample-to-chunk atom.
- Type. A long integer that specifies the type of the data in this sample-to-chunk atom (defined by the `'stsc'` atom type).
- Version. A 1-byte specification of the version of this sample-to-chunk atom.
- Flags. Three bytes of space for future flags associated with this sample-to-chunk atom.

- Number of entries. The number of entries in the sample-to-chunk table.
- Sample-to-chunk table. Figure 4-31 shows the structure of a sample-to-chunk table. Each sample-to-chunk atom contains such a table, which identifies the chunk for each sample in a media. Each entry in the table contains a first chunk field, a samples per chunk field, and a sample description ID field. From this information, you can ascertain where samples reside in the media data.

Figure 4-31 The layout of a sample-to-chunk table

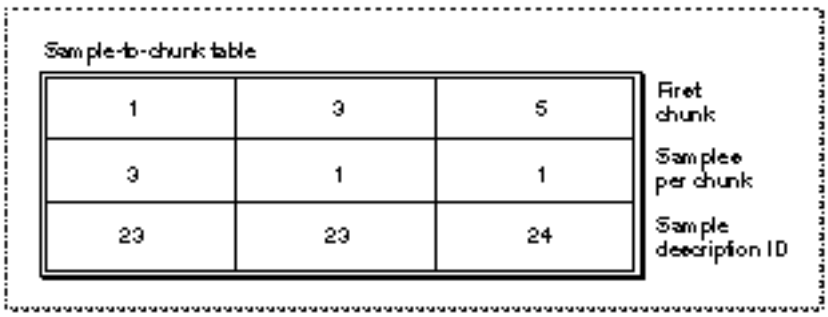


You define a sample-to-chunk table by specifying these elements:

- First chunk. The first chunk number using this table entry.
- Samples per chunk. The number of samples in each chunk.
- Sample description ID. The identification number associated with the sample description containing the sample. For details on sample description atoms, see “Sample Description Atoms,” which begins on page 4-35.

Figure 4-32 shows an example of a sample-to-chunk table that is based on the data stream shown in Figure 4-22.

Figure 4-32 An example of a sample-to-chunk table



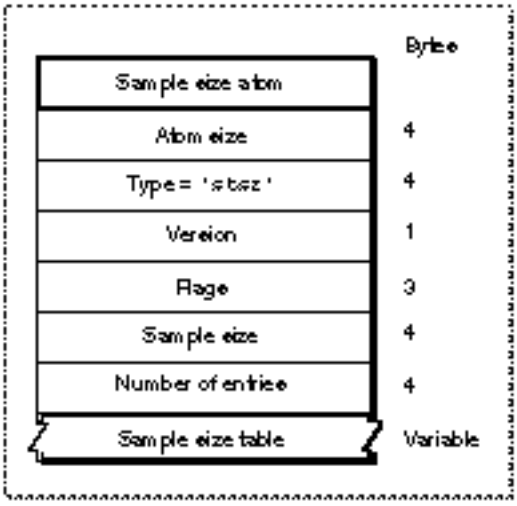
Each table entry corresponds to a set of consecutive chunks, each of which contains the same number of samples. Furthermore, each of the samples in these chunks must use the same sample description (see “Sample Description Atoms,” which begins on page 4-35). Whenever the number of samples per chunk or the sample description changes, you must create a new table entry. If all the chunks have the same number of samples per chunk and use the same sample description, this table has one entry.

Sample Size Atoms

You use sample size atoms to identify the size of each sample in the media.

Sample size atoms have an atom type of `'stsz'`. The sample size atom (shown in Figure 4-33) contains sample size information.

Figure 4-33 The layout of a sample size atom



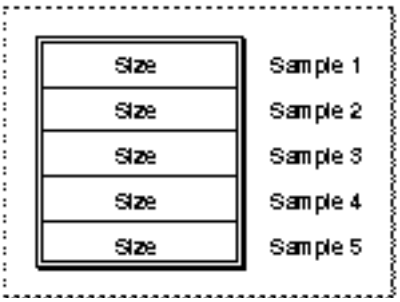
You define a sample size atom by specifying these elements:

- Size. A long integer that specifies the number of bytes in this sample size atom.
- Type. A long integer that specifies the type (of atom type `'stsz'`) of the data in this sample size atom.
- Version. A 1-byte specification of the version number of this sample size atom.
- Flags. Three bytes of space for future flags associated with the data in this sample size atom.
- Sample size. The number of bytes in the samples in the sample size table field. If all the samples are the same size, the sample size field of this atom indicates the size of all the samples. If this field is set to 0, then the samples have different sizes, and those sizes are stored in the sample size table.

- Number of entries. The number of entries in the sample size table contained in the sample size table field of this atom.
- Sample size table. The sample size table, which contains the sample size information. A sample size table contains an entry for every sample. Each table entry contains a size field. There is one table entry for each sample in the media. The table is indexed by sample number—the first entry corresponds to the first sample, the second entry is for the second sample, and so on. The size field contains the size, in bytes, of the sample in question.

Figure 4-34 shows the sample size table for the data stream represented in Figure 4-22 on page 4-33.

Figure 4-34 An example of a sample size table

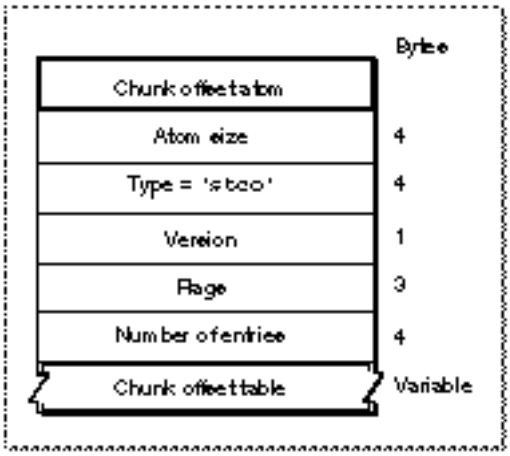


Chunk Offset Atoms

Chunk offset atoms identify the location of each chunk of data in the media's data stream.

Chunk offset atoms have an atom type of `'stco'`. The chunk offset atom (shown in Figure 4-35) contains a table of offset information.

Figure 4-35 The layout of a chunk offset atom

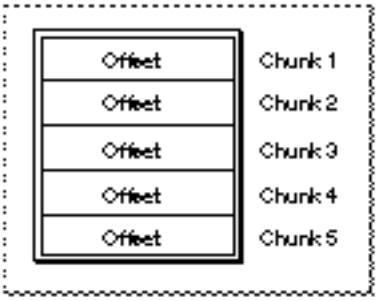


You define a chunk offset atom by specifying these elements:

- Size. A long integer that specifies the number of bytes in this chunk offset atom.
- Type. A long integer that specifies the type of the data in this chunk offset atom (defined by the atom type `'stco'`).
- Version. A 1-byte specification of the version of this chunk offset atom.
- Flags. A 3-byte space for future flags associated with this chunk offset atom.
- Number of entries. A long integer that specifies the number of entries in the chunk offset table.
- Chunk offset table. The chunk offset table, which consists of a number of offset fields. Each entry in the chunk offset table contains an offset field. There is one table entry for each chunk in the media. The table is indexed by chunk number—the first table entry corresponds to the first chunk, the second table entry is for the second chunk, and so on. The offset field contains the byte offset from the beginning of the data stream to the chunk.

Figure 4-36 shows an example of the chunk offset table for the data stream represented by Figure 4-22 on page 4-33.

Figure 4-36 An example of a chunk offset table

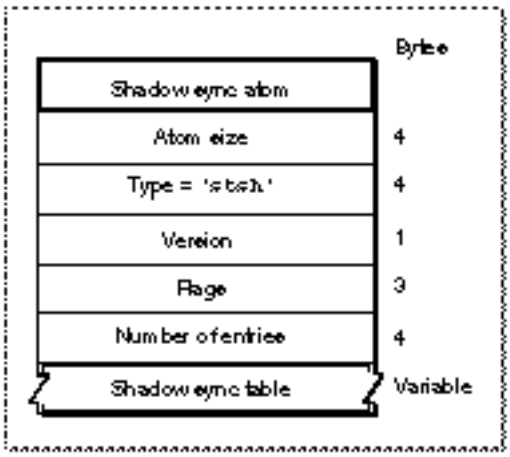


Shadow Sync Atoms

Shadow sync atoms contain self-contained samples that are alternates for existing frame difference samples. Shadow sync atoms are used to optimize random access operations on a movie. Scrubbing is an example of such a random access operation. These atoms are used to enhance playback performance. See the chapter "Movie Toolbox" in this book for details on the `SetMediaShadowSync` and `GetMediaShadowSync` functions, which allow you to create an association between a frame difference sample and a sync sample.

Figure 4-37 shows the layout of a shadow sync atom. Shadow sync atoms have an atom type of `'stsh'`. Each shadow sync atom contains a table with a frame difference number and a sync sample number.

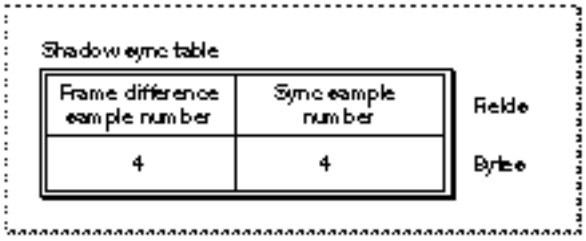
Figure 4-37 The layout of a shadow sync atom



You define a shadow sync atom by specifying these elements:

- Size. A long integer that specifies the number of bytes in this shadow sync atom.
- Type. A long integer that specifies the type (defined by the atom type `'stsh'`) of the data in this shadow sync atom.
- Version. A 1-byte specification of the version number of this shadow sync atom.
- Flags. Three bytes of space for future flags.
- Number of entries. A long integer that specifies how many entries in the shadow sync table are listed in the shadow sync table field of this atom.
- Shadow sync table. The shadow sync table, which contains the shadow sync information. The shadow sync table is shown in Figure 4-38.

Figure 4-38 The layout of a shadow sync table



A shadow sync table contains a frame difference sample number and a sync sample number.

Using Media Information Atoms

This section presents examples using the atoms just described. These examples are intended to help you understand the relationships between these atoms. The first example, “Finding a Sample,” describes the steps that the video media handler uses to find the sample that contains the media data for a particular time in a media. The second example, “Finding a Key Frame,” describes the steps that the video media handler uses to find an appropriate key frame for a specific time in a movie.

Finding a Sample

When it displays a movie or track, QuickTime tells the appropriate media handler to access the media data for a particular time. The media handler must correctly interpret the data stream to retrieve the requested data. In the case of video media, the media handler traverses several atoms to find the location and size of a sample for a given media time. The media handler does the following:

1. Determines the time in the media time coordinate system.
2. Examines the time-to-sample atom to determine the sample number that contains the data for the specified time.
3. Scans the sample-to-chunk atom to discover which chunk contains the sample in question.
4. Extracts the offset to the chunk from the chunk offset atom.
5. Finds the offset within the chunk by using the sample size atom.

Finding a Key Frame

Finding a key frame for a specified time in a movie is slightly more complicated than finding a sample for a specified time. The media handler must use the sync sample atom and the time-to-sample atom together in order to find a key frame. The media handler does the following:

1. Examines the time-to-sample atom to determine the sample number that contains the data for the specified time.
2. Scans the sync sample atom to find the key frame that precedes the sample number chosen in step 1.
3. Scans the sample-to-chunk atom to discover which chunk contains the key frame.
4. Extracts the offset to the chunk from the chunk offset atom.
5. Finds the offset within the chunk by using the sample size atom.

This chapter has described the format of QuickTime movie resources for those developers who need to know about the content of movie resources. The knowledge you have gained about movie resources should help you in the creation of movies on other computers and in the process of importing them to the Macintosh environment, or in the interpretation of QuickTime movies on other types of computers.

Glossary

action One of many integer constants used by QuickTime movie controller components in the `MCDoAction` function. Applications that include action filters may receive any of these actions.

active movie segment A portion of a QuickTime movie that is to be used for playback. By default, the active segment is set to the entire movie. You can change the active segment of a movie by using the Movie Toolbox.

active source rectangle The portion of the **maximum source rectangle** that contains active video that can be digitized by a video digitizer component.

aliasing The result of sampling a signal at less than twice its natural frequency. Aliasing causes data to be lost in the conversion that occurs when resampling an existing signal at more than twice its natural frequency.

alpha channel The portion of each display pixel that represents the blending of video and graphical image data for a video digitizer component.

alternate group A collection of movie **tracks** that contain alternate data for one another. The Movie Toolbox chooses one track from the group to be used when the movie is played. The choice may be based on such considerations as quality or language.

anti-aliasing The process of sampling a signal at more than twice its natural frequency to ensure that **aliasing** artifacts do not occur.

area of interest The portion of a test image that is to be displayed in the standard image-compression dialog box.

atom The basic unit of data in a movie resource. There are a number of different atom types, including movie atoms, track atoms, and media atoms. There are two varieties of atoms: container atoms, which contain other atoms, and leaf atoms, which do not contain any other atoms.

attached controller A movie controller with an attached movie.

automatic key frame A key frame that is inserted automatically by the Image Compression Manager when it detects a scene change. When performing temporal compression, the Image Compression Manager looks for frames that have changed more than 90 percent since the previous frame. If such a change occurs, the Image Compression Manager assumes a scene change and inserts a key frame. A **key frame** allows fast random access and reverse play in addition to efficient compression and picture quality of the frame.

badge A visual element in a movie's display that distinguishes a movie from a static image. The movie controller component supplied by Apple supports badges.

band A horizontal strip from an image. The Image Compression Manager may break an image into bands if a compressor or decompressor component cannot handle an entire image at once.

base media handler component A component that handles most of the duties that must be performed by all **media handlers.** See also **derived media handler component.**

black level The degree of blackness in an image. This is a common setting on a video digitizer. The highest setting will produce an all-black image whereas the lowest setting will yield very little, if any, black even with black objects in the scene. Black level is an important digitization setting since it can be adjusted so that there is little or no noise in an image.

blend matte A pixel map that defines the blending of video and digital data for a video digitizer component. The value of each pixel in the pixel map governs the relative intensity of the video data for the corresponding pixel in the result image.

callback event A scheduled invocation of a Movie Toolbox **callback function.** Applications establish the criteria that determine when the callback function is to be invoked. When those criteria are met, the Movie Toolbox invokes the callback function.

callback function An application-defined function that is invoked at a specified time or based on specified criteria. These callback functions are data-loading functions, data-unloading functions, completion functions, and progress functions. See also **callback event.**

chunk In the movie resource formats, a collection of sample data in a media. Chunks allow optimized data access. A chunk may contain one or more samples. Chunks in a media may have different sizes and the samples within a chunk may have different sizes. In the Sound Manager, a chunk may refer to a collection of sampled sound and definitions of the characteristics of sampled sound and other relevant details about the sound.

clipped movie boundary region The region that is clipped by the Movie Toolbox. This region combines the union of all track movie boundary regions for a movie, which is the movie's **movie boundary region,** with the movie's **movie clipping region,** which defines the portion of the movie boundary region that is to be used.

clock component A **component** that supplies basic time information to its clients. Clock components have a **component type** value of `'clok'`.

color ramps Images in which the shading goes from light to dark in smooth increments.

component A software entity, managed by the Component Manager, that provides a defined set of services to its clients. Examples include clock components, movie controller components, and image compressor components.

component instance A channel of communication between a **component** and its client.

component subtype An element in the classification hierarchy used by the Component Manager to define the services provided by a **component.** Within a **component type,** the component subtype provides additional information about the component. For example, image compressor components all have the same component type value; the component subtype value indicates the compression algorithm implemented by the component.

component type An element in the classification hierarchy used by the Component Manager to define the services provided by a **component.** The component type value indicates the type of services provided by the component. For example, all image compressor components have a component type value of `'imco'`. See also **component subtype.**

compressor component A general term used to refer to both **image compressor components** and **image decompressor components.**

connection A channel of communication between a **component** and its client. A **component instance** is used to identify the connection.

container atom A QuickTime atom that contains other atoms, possibly including other container atoms. Examples of container atoms are track atoms and edit atoms. Compare **leaf atom.**

controller boundary rectangle The rectangle that completely encloses a movie controller. If the controller is attached to its movie, the rectangle also encloses the movie image.

controller boundary region The region occupied by a movie controller. If the controller is attached to its movie, the region also includes the movie image.

controller clipping region The clipping region of a movie controller. Only the portion of the controller and its movie that lies within the clipping region is visible to the user.

controller window region The portion of a movie controller and its movie that is visible to the user.

cover function An application-defined function that is called by the Movie Toolbox whenever a movie covers a portion of the screen or reveals a portion of the screen that was previously hidden by the movie.

current error One of two error values maintained by the Movie Toolbox. The current error value is updated by every Movie Toolbox function. The other error value, the **sticky error,** is updated only when an application directs the Movie Toolbox to do so.

current selection A portion of a QuickTime movie that has been selected for a cut, copy, or paste operation.

current time The time value that represents the point of a QuickTime movie that is currently playing or would be playing if the movie had a nonzero rate value.

data dependency An aspect of image compression in which compression ratios are highly dependent on the image content. Using an algorithm with a high degree of data dependency, an image of a crowd at a football game (which contains a lot of detail) may produce a very small compression ratio, whereas an image of a blue sky (which consists mostly of constant colors and intensities) may produce a very high compression ratio.

data handler A piece of software that is responsible for reading and writing a media's data. The data handler provides data input and output services to the media's **media handler.**

data reference A reference to a media's data.

derived media handler component A component that allows the Movie Toolbox to access the data in a media. Derived media handler components isolate the Movie Toolbox from the details of how or where a particular media is stored. This not only frees the Movie Toolbox from reading and writing media data, but also makes QuickTime extensible to new data formats and storage devices. These components are referred to as *derived* components because they rely on the services of a common base media handler component, which is supplied by Apple. See also **base media handler component.**

detached controller A movie controller component that is separate from its associated movie.

digitizer rectangle The portion of the **active source rectangle** that you want to capture and convert with a video digitizer component.

display coordinate system The QuickDraw graphics world, which can be used to display QuickTime movies, as opposed to the movie's **time coordinate system,** which defines the basic time unit for each of the movie's tracks.

dithering A technique used to improve picture quality when you are attempting to display an image that exists at a higher bit-depth representation on a lower bit-depth device. For example, you might want to dither a 24 bits per pixel image for display on an 8-bit screen.

duration A time interval. Durations are time values that are interpreted as spans of time, rather than as points in time.

edit state Information defining the current state of a movie or track with respect to an edit session. The Movie Toolbox uses edit states to support its undo facilities.

fixed point A point that uses fixed-point numbers to represent its coordinates. The Movie Toolbox uses fixed points to provide greater display precision for graphical and image data.

fixed rectangle A rectangle that uses **fixed points** to represent its vertices. The Movie Toolbox uses fixed rectangles to provide greater display precision.

flattening The process of copying all of the original data referred to by reference in QuickTime tracks into a QuickTime movie file. This can also be called *resolving references.* Flattening is used to bring in all of the data that may be referred to from multiple files after QuickTime editing is complete. It makes a QuickTime movie stand-alone—that is, it can be played on any system without requiring any additional QuickTime movie files or tracks, even if the original file referenced hundreds of files. The flattening operation is essential if QuickTime movies are to be used with CD/ROM discs.

frame A single image in a **sequence** of images.

frame differencing A form of temporal compression that involves examining redundancies between adjacent frames in a moving image sequence. Frame differencing can improve compression ratios considerably for a video sequence.

frame rate The rate at which a movie is displayed—that is, the number of frames per second that are actually being displayed. In QuickTime the frame rate at which a movie was recorded may be different from the frame rate at which it is displayed. On very fast machines, the playback frame rate may be faster than the record frame rate; on slow machines, the playback frame rate may be slower than the record frame rate. Frame rates may be fractional.

genlock A circuit that locks the frequency of an internal clock to an external timing source. This term is used to refer to the ability of a video digitizer to rely on external clocking.

hue value A setting that is similar to the tint control on a television. Hue value can be specified in degrees with complementary colors set 180˚ apart (red is 0˚, green is +120˚, and blue is –120˚). Video digitizer components support hue values that range from 0 (–180˚ shift in hue) to 65,535 (+179˚ shift in hue), where 32,767 represents a 0˚ shift in hue. Hue value is set with the video digitizer component's `VDSetHue` function.

identity matrix A **transformation matrix** that specifies no change in the coordinates of the source image. The resulting image corresponds exactly to the source image.

image compressor component A **component** that provides image-compression services. Image compressor components have a **component type** of `'imco'`.

image decompressor component A **component** that provides image-decompression services. Image decompressor components have a **component type** value of `'imdc'`.

image sequence A series of visual representations usually represented by video over time. Image sequences may also be generated synthetically, such as from an animation sequence.

interesting time A time value in a movie, track, or media that meets certain search criteria. You specify the search criteria in the Movie Toolbox. The Movie Toolbox then scans the movie, track, or media and locates time values that meet those search criteria.

interlacing A video mode that updates half the scan lines on one pass and goes through the second half during the next pass.

interleaving A technique in which sound and video data are alternated in small pieces, so the data can be read off disk as it is needed. Interleaving allows for movies of almost any length with little delay on startup.

intraframe coding A process that compresses only a single frame. It does not require looking at adjacent frames in time to achieve compression, but allows fast random access and reverse play.

Joint Photographic Experts Group (JPEG) Refers to an international standard for compressing still images. This standard supplies the algorithm for image compression. The version of JPEG supplied with QuickTime complies with the baseline International Standards Organization (ISO) standard bitstream, version 9R9. This algorithm is best suited for use with natural images.

JPEG See **Joint Photographic Experts Group.**

key color A color in a destination image that is replaced with video data by a video digitizer component. Key colors represent one technique for selectively displaying video on a computer display. Other techniques include the use of **alpha channels** and **blend mattes.**

key frame A sample in a sequence of temporally compressed samples that does not rely on other samples in the sequence for any of its information. Key frames are placed into temporally compressed sequences at a frequency that is determined by the **key frame rate.** Typically, the term *key frame* is used with respect to temporally compressed sequences of image data. See also **sync sample.**

key frame rate The frequency with which **key frames** are placed into temporally compressed data sequences.

layer A mechanism for prioritizing the tracks in a movie. When it plays a movie, the Movie Toolbox displays the movie's tracks according to their layer—tracks with lower layer numbers are displayed first; tracks with higher layer numbers are displayed over those tracks.

leaf atom A QuickTime atom that contains no other atoms. A leaf atom, however, may contain a table. An example of a leaf atom is an edit list atom. The edit list atom contains the edit list table. Compare **container atom.**

lossless compression A compression scheme that preserves all of the original data.

lossy compression A compression scheme that does not preserve the data precisely; some data is lost, and it cannot be recovered after compression. Most lossy schemes try to compress the data as much as possible, without decreasing the image quality in a noticeable way.

mask region A 1-bit-deep region that defines how an image is to be displayed in the destination coordinate system. For example, during decompression the Image Compression Manager displays only those pixels in the source image that correspond to bits in the mask region that are set to 1. Mask regions must be defined in the destination coordinate system.

master clock component A movie's clock component.

matrix See **transformation matrix.**

matte See **blend matte, track matte.**

maximum source rectangle A rectangle representing the maximum source area that a video digitizer component can grab. This rectangle usually encompasses both the vertical and horizontal blanking areas.

media A Movie Toolbox data structure that contains information that describes the data for a track in a movie. Note that a media does not contain its data; rather, a media contains a reference to its data, which may be stored on disk, CD-ROM disc, or any other mass storage device.

media handler A piece of software that is responsible for mapping from the movie's time coordinate system to the media's time coordinate system. The media handler also interprets the media's data. The **data handler** for the media is responsible for reading and writing the media's data. See also **base media handler component, derived media handler component.**

media information Control information about a media's data that is stored in the media structure by the appropriate **media handler.**

movie A set of time-based data that is managed by the Movie Toolbox. A QuickTime movie may contain sound, video, animation, laboratory results, financial data, or a combination of any of these types of time-based data. A QuickTime movie contains one or more **tracks;** each track represents a single data stream in the movie.

movie boundary region A region that describes the area occupied by a movie in the movie coordinate system, before the movie has been clipped by the **movie clipping region.** A movie's boundary region is built up from the **track movie boundary regions** for each of the movie's **tracks.**

movie box A rectangle that completely encloses the **movie display boundary region.** The movie box is defined in the display coordinate system.

movie clipping region The clipping region of a movie in the movie's coordinate system. The Movie Toolbox applies the movie's clipping region to the **movie boundary region** to obtain a clipped movie boundary region. Only that portion of the movie that lies in the clipped movie boundary region is then transformed into an image in the display coordinate system.

movie controller component A component that manages movie controllers, which present a user interface for playing and editing movies.

movie data exchange component A component that allows applications to move various types of data into and out of a QuickTime movie. The two types of data exchange components, which provide data conversion services to and from standard QuickTime movie data formats, are the **movie import component** and the **movie export component.**

movie data export component A component that converts QuickTime movie data into other formats.

movie data import component A component that converts other data formats into QuickTime movie data format.

movie display boundary region A region that describes the display area occupied by a movie in the display coordinate system, before the movie has been clipped by the **movie display clipping region.**

movie display clipping region The clipping region of a movie in the display coordinate system. Only that portion of the movie that lies in the clipping region is visible to the user. The Movie Toolbox applies the movie's display clipping region to the **movie display boundary region** to obtain the visible image.

movie file A QuickTime file that stores all information about the movie in a Macintosh resource, and stores all the associated data for the movie separately. The resource is stored in the resource fork, and the data in the data fork. Most QuickTime movies are stored in files with double forks. Compare **single-fork movie file.**

movie poster A single visual image representing a QuickTime movie. You specify a poster as a point in time in the movie and specify the tracks that are to be used to constitute the poster image.

movie preview A short dynamic representation of a QuickTime movie. Movie previews typically last no more than 3 to 5 seconds, and they should give the user some idea of what the movie contains. You define a movie preview by specifying its start time, duration, and its tracks.

movie resource One of several data structures that provide the medium of exchange for movie data between applications on a Macintosh computer and between computers, even computers of different types.

National Television System Committee (NTSC) Refers to the color-encoding method adopted by the committee in 1953. This standard was the first monochrome-compatible, simultaneous color transmission system used for public broadcasting. This method is used widely in the United States.

NTSC See **National Television System Committee.**

offset-binary encoding A method of digitally encoding sound that represents the range of amplitude values as an unsigned number, with the midpoint of the range representing silence. For example, an 8-bit sound sample stored in offset-binary format would contain sample values ranging from 0 to 255, with a value of 128 specifying silence (no amplitude). Samples in Macintosh sound resources are stored in offset-binary form. Compare **twos-complement encoding.**

PAL See **Phase Alternation Line.**

palindrome looping Running a movie in a circular fashion from beginning to end and end to beginning, alternating forward and backward. Looping must also be enabled in order for palindrome looping to take effect.

Phase Alternation Line (PAL) A color-encoding system used widely in Europe, in which one of the subcarrier phases derived from the color burst is inverted in phase from one line to the next. This technique minimizes hue errors that may result during color video transmission. Sometimes called *Phase Alternating Line*.

phase-locked loop (PLL) A piece of hardware that synchronizes itself to an input signal—for example, a video digitizer card that synchronizes to an incoming video source. The video digitizer component's `VDSetPLLFilterType` function allows applications to specify which phase-locked loop is to be active.

playback quality A relative measure of the fidelity of a track in a QuickTime movie. You can control the playback (or language) quality of a movie during movie playback. The Movie Toolbox chooses tracks from **alternate groups** that most closely correspond to the display quality you desire. In this manner you can create a single movie that can take advantage of the hardware configurations of different computer systems during playback.

PLL See **phase-locked loop.**

preferred rate The default playback rate for a QuickTime movie.

preferred volume The default sound volume for a QuickTime movie.

preroll A technique for improving movie playback performance. This technique is used when prerolling a movie. The Movie Toolbox

informs the movie's **media handlers** that the movie is about to be played. The media handlers can then load the appropriate movie data. In this manner, the movie can play smoothly from the start.

preview A short, potentially dynamic, visual representation of the contents of a file. The Standard File Package can use file previews in file dialog boxes to give the user a visual cue about a file's contents.

preview component A component used by the Movie Toolbox's standard file preview functions to display and create visual previews for files. Previews usually consist of a single image, but they may contain many kinds of data, including sound. In QuickTime, the Movie Toolbox is the primary client of preview components. Rarely, if ever, do applications call preview components directly.

progress function An application-defined function that is invoked by the Movie Toolbox or the Image Compression Manager. You can use these functions to track the progress of time-consuming activities, and thereby keep the user informed about that progress.

rate A value that specifies the pace at which time passes for a **time base.** A time base's rate is multiplied by the time scale to obtain the number of **time units** that pass per second. For example, consider a time base that operates in a time coordinate system that has a time scale of 60. If that time base has a rate of 1, 60 time units are processed per second. If the rate is set to 1/2, 30 time units pass per second. If the rate is 2, 120 time units pass per second.

sample A single element of a sequence of time-ordered data.

sample number A number that identifies the sample with data for a specified time.

saturation value A setting that controls color intensity. For example, at high saturation levels, red appears to be red; at low saturation, red appears pink. Valid saturation values range from 0 to 65,535, where 0 is the minimum saturation value and 65,535 specifies maximum saturation. Saturation value is set with the video digitizer component's `VDSetSaturation` function.

SECAM See **Systeme Electronique Couleur avec Memoire.**

selection duration A time value that specifies the duration of the **current selection** of a movie.

selection time A time value that specifies the starting point of the **current selection** of a movie.

sequence A series of images that may be compressed as a sequence. To do this, the images must share an image description structure. In other words, each image or **frame** in the sequence must have the same compressor type, pixel depth, color lookup table, and boundary dimensions.

sequence grabber channel component A component that manipulates captured data for **sequence grabber components.**

sequence grabber component A component that allows applications to obtain digitized data from sources that are external to a Macintosh computer. For example, you can use a sequence grabber component to record video data from a **video digitizer component.** Your application can then request that the sequence grabber store the captured video data in a QuickTime movie. In this manner you can acquire movie data from various sources that can augment the movie data you create by other means, such as computer animation. You can also use sequence grabber components to obtain and display data from external sources, without saving the captured data in a movie.

sequence grabber panel component A component that allows sequence grabber components to obtain configuration information from the user for a particular **sequence grabber channel component.** An application never calls a sequence grabber panel component directly; application developers use panel components only by calling the **sequence grabber component.**

shadow sync sample A self-contained sample that is an alternate for an already existing frame difference sample. During certain random access operations, a shadow sync sample is used instead of a normal key frame, which may be very far away from the desired frame. See also **frame differencing.**

single-fork movie file A QuickTime movie file that stores both the movie data and the movie resource in the data fork of the movie file. You can use single-fork movie files to ease the exchange of QuickTime movie data between Macintosh computers and other computer systems. Compare **movie file.**

spatial compression Image compression that is performed within the context of a single **frame.** This compression technique takes advantage of redundancy in the image to reduce the amount of data required to accurately represent the image. Compare **temporal compression.**

standard image-compression dialog component A component that provides a consistent user interface for selecting parameters that govern compression of an image or image sequence and then manages the compression operation.

sticky error One of two error values maintained by the Movie Toolbox. The sticky error is updated only when an application directs the Movie Toolbox to do so. The other error value, the **current error,** is updated by every Movie Toolbox function.

s-video A video format in which color and brightness information are encoded as separate signals. The s-video format is component video as opposed to composite video, which is the NTSC standard.

sync sample A sample that does not rely on preceding frames for content. See also **key frame.**

Systeme Electronique Couleur avec Memoire (SECAM) Sequential Color With Memory; refers to a color-encoding system in which the red and blue color-difference information is transmitted on alternate lines, requiring a one-line memory in order to decode green information.

tearing The effect you obtain if you redraw the screen from the buffer while the buffer is only half updated, so that you get one half of one image and one half of another on a single raster scan.

temporal compression Image compression that is performed between **frames** in a sequence. This compression technique takes advantage of redundancy between adjacent frames in a sequence to reduce the amount of data that is required to accurately represent each frame in the sequence. Sequences that have been temporally compressed typically contain **key frames** at regular intervals. Compare **spatial compression.**

thumbnail picture A picture that can be created from an existing image that is stored as a pixel map, a picture, or a picture file. A thumbnail picture is useful for creating small representative images of a source image and in previews for files that contain image data.

time base A set of values that define the time basis for an entity, such as a QuickTime movie. A time base consists of a **time coordinate system** (that is, a **time scale** and a **duration**) along with a rate value. The rate value specifies the speed with which time passes for the time base.

time coordinate system A set of values that defines the context for a **time base.** A time coordinate system consists of a **time scale** and a **duration.** Together, these values define the coordinate system in which a **time value** or a time base has meaning.

time scale The number of **time units** that pass per second in a **time coordinate system.** A time coordinate system that measures time in sixtieths of a second, for example, has a time scale of 60.

time unit The basic unit of measure for time in a time coordinate system. The value of the time unit for a time coordinate system is represented by the formula (1/time scale) seconds. A time coordinate system that has a time scale of 60 measures time in terms of sixtieths of a second.

time value A value that specifies a number of time units in a **time coordinate system.** A time value may contain information about a point in time or about a **duration.**

track A Movie Toolbox data structure that represents a single data stream in a QuickTime **movie.** A movie may contain one or more tracks. Each track is independent of other tracks in the

movie and represents its own data stream. Each track has a corresponding **media.** The media describes the data for the track.

track boundary region A region that describes the area occupied by a track in the track's coordinate system. The Movie Toolbox obtains this region by applying the **track clipping region** and the **track matte** to the visual image contained in the **track rectangle.**

track clipping region The clipping region of a track in the track's coordinate system. The Movie Toolbox applies the track's clipping region and the **track matte** to the image contained in the **track rectangle** to obtain the **track boundary region.** Only that portion of the track that lies in the track boundary region is then transformed into an image in the movie coordinate system.

track height The height, in pixels, of the **track rectangle.**

track matte A pixel map that defines the blending of track visual data. The value of each pixel in the pixel map governs the relative intensity of the track data for the corresponding pixel in the result image. The Movie Toolbox applies the track matte, along with the **track clipping region,** to the image contained in the **track rectangle** to obtain the **track boundary region.**

track movie boundary region A region that describes the area occupied by a track in the movie coordinate system, before the movie has been clipped by the **movie clipping region.** The **movie boundary region** is built up from the track movie boundary regions for each of the movie's **tracks.**

track offset The blank space that represents the intervening time between the beginning of a movie and the beginning of a track's data. In an audio track, the blank space translates to silence; in a video track, the blank space generates no visual image. All of the tracks in a movie use the movie's time coordinate system. That is, the movie's time scale defines the basic time unit for each of the movie's tracks. Each track begins at the beginning of the movie, but the track's data might not begin until some time value other than 0.

track rectangle A rectangle that completely encloses the visual representation of a track in a QuickTime movie. The width of this rectangle in pixels is referred to as the **track width;** the height, as the **track height.**

track width The width, in pixels, of the track rectangle.

transformation matrix A 3-by-3 matrix that defines how to map points from one coordinate space into another coordinate space.

twos-complement encoding A system for digitally encoding sound that stores the amplitude values as a signed number—silence is represented by a sample with a value of 0. For example, with 8-bit sound samples, twos-complement values would range from –128 to 127, with 0 meaning silence. The Audio Interchange File Format (AIFF) used by the Sound Manager stores samples in twos-complement form. Compare **offset-binary encoding.**

user data Auxiliary data that your application can store in a QuickTime movie, track, or media structure. The user data is stored in a **user data list;** items in the list are referred to as **user data items.** Examples of user data include a copyright, date of creation, name of a movie's director, and special hardware and software requirements.

user data item A single element in a **user data list.**

user data list The collection of **user data** for a QuickTime movie, track, or media. Each element in the user data list is referred to as a **user data item.**

vertical blanking rectangle A rectangle that defines a portion of the input video signal that is devoted to vertical blanking. This rectangle occupies lines 10 through 19 of the input signal. Broadcast video sources may use this portion of the input signal for closed captioning, teletext, and other nonvideo information. Note that the blanking rectangle cannot be contained in the **maximum source rectangle.**

video digitizer component A component that provides an interface for obtaining digitized video from an analog video source. The typical client of a video digitizer component is a sequence grabber component, which uses the services of video digitizer components to create a very simple interface for making and previewing movies. Video digitizer components can also operate independently, placing live video into a window.

white level The degree of whiteness in an image. It is a common video digitizer setting.

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W, X, Y, Z

This Apple manual was written, edited, and composed on a desktop publishing system using Apple Macintosh computers and FrameMaker software. Proof pages were created on an Apple LaserWriter IINTX printer. Final page negatives were output directly from text files on an AGFA ProSet 9800 imagesetter. Line art was created using Adobe™ Illustrator. PostScript™, the page-description language for the LaserWriter, was developed by Adobe Systems Incorporated.

Text type is Palatino® and display type is Helvetica®. Bullets are ITC Zapf Dingbats®. Some elements, such as program listings, are set in Apple Courier.

WRITERS
Doug Engfer and Patria Brown

DEVELOPMENTAL EDITOR
Sue Factor

ILLUSTRATOR
Ruth Anderson

PRODUCTION EDITORS
Pat Christenson, Josephine Manuele

PROJECT MANAGER
Patricia Eastman

COVER DESIGNER
Barbara Smyth

Special thanks to Jim Batson, Julie Callahan, Sean Callahan, Ken Doyle, Peter Hoddie, Sanborn Hodgkins, Mark Krueger, Bruce Leak, Kip Olson, and Laurel Rezeau.

Acknowledgments to Rita Brennan, Eric Chan, Mike Dodd, Bill Guschwan, Eric Hoffert, Miki Lee, Guillermo Ortiz, Martha Steffen, John Wang, Gary Woodcock, Bill Wright, and the entire *Inside Macintosh* team.