QuickTime 3 Reference

For Macintosh and Windows

Apple Computer, Inc.
Technical Publications
April, 1998

Contents

Figures, Tables, and Listings

About This Book

This reference book is designed and written as a delta guide to QuickTime 3 for Macintosh and Windows. It is aimed primarily at Windows and Macintosh developers who are working with or building applications using QuickTime 3. This guide describes all the features added or changed in QuickTime since version 1.5, and therefore supersedes all existing documentation for QuickTime software releases 1.6.1, 2.0, 2.1, and 2.5.

The original QuickTime-related *Inside Macintosh* books only documented QuickTime for Macintosh through the QuickTime 1.5 software release. With this updated reference guide, your main source for information on programming in QuickTime is now the combination of this book plus these others, all of which are included in the QuickTime 3 Software Developers Kit (SDK):

- *Inside Macintosh: QuickTime*
- *Inside Macintosh: QuickTime Components*
- *Mac OS for QuickTime Programmers*
- *QuickTime 3 for Windows Programmers*
- *QuickTime Music Architecture*
- *Programming with QuickTime Sprites*

A new book describing QuickTime file formats has also been added to the QuickTime for Macintosh documentation suite. *QuickTime File Format Specification, May 1996,* supersedes Chapter 4 of *Inside Macintosh: QuickTime,* ”Movie Resource Formats.” In addition, Appendix B of this book, “QuickTime File Formats,” provides new information that has not yet been incorporated into the file format specification book.

Many chapters in this book update specific chapters in the *Inside Macintosh* books. Within these update chapters, the section headings correspond to sections found in the *Inside Macintosh* books, wherever possible. However, the update chapters in this book contain only the changed information; you must refer to the *Inside Macintosh* books for all unchanged features. This book also contains chapters that describe completely new areas of functionality that were not documented in the *Inside Macintosh* books.

What's New in QuickTime 3

QuickTime 3 contains many features that are new or improved from earlier releases. This section gives you a brief overview of them.

Overview of New Features Documented

This *QuickTime 3 Reference* includes information about the following new features:

- Support for access keys, which control access to encrypted data. For an introduction to this feature, see "Support for Access Keys" (page 50).
- Support for compressed movie resources. For an introduction to this feature, see "Compressed Movie Resources" (page 51). For information about the storage format for compressed movie resources, see "Compressed Movie Resources" (page 989).
- Support for DV video. Compressor and decompressor components for DV video are described in (page 230). A decompressor component for DV audio is described on page (page 577). Data exchange components for DV video are described in "DV Video Import and Export Components" (page 352). For a description of the file format for DV video data, see "DV Files" (page 989).
- The ability to specify the size of the image buffer for image compressor or decompressor components. For an introduction to this feature, see "Specifying the Size of an Image Buffer" (page 231).
- The ability to store captured data in multiple files. For an introduction to this feature, see "Storing Captured Data in Multiple Files" (page 280) and "Capturing to Multiple Files" (page 306).
- The ability to export data from sources other than movies. For an introduction to this feature, see "Exporting Data from Sources Other Than Movies" (page 353).
- Simpler implementation of movie data export components. For an introduction to this feature, see "Implementing Movie Data Export Components" (page 366).

- Support for video output devices other than monitors. This feature is described in Chapter 18, “Video Output Components.”
- Support for adding new compressor and decompressor components is described in Chapter 23, “Data Codec Components.”
- The standard sound component, which lets applications display a dialog for configuring their sound settings and get the settings that the user provides, is described in Chapter 20, “Standard Sound Dialog Component.”
- The base image compressor, a component that performs tasks for image decompressor components, including the scheduling of asynchronous decompression operations, is described in Chapter 21, “The Base Image Decompressor.”
- Support for real-time video effects is described in Chapter 25, “QuickTime Video Effects.”
- Support for progressive downloads. For an introduction to this feature, see “Support for Progressive Downloads” (page 51).
- Support for MIME types in graphics and movie import components. For an introduction to MIME type support for movie import components, see “Getting a List of Supported MIME Types” (page 364). For information about MIME type support for graphics import components, see “MIME Type List” (page 492).
- Support for aliases to components. For an introduction to this feature, see “Component Aliases” (page 173).
- Functions that permit decompression at interrupt time. For an introduction to this feature, see “Compressing or Decompressing During Interrupt Time” (page 750).
- Support for vector-based graphics. For an introduction to these features, see Chapter 24, “QuickTime Vectors.”

Format of an Update Chapter

The update chapters in this book follow a standard structure where possible. Each chapter begins with an overview of the new features. Some of the chapters give programming examples, then all the constants, data types, and functions are described in a reference section. For example, Chapter 1, “Movie Toolbox,” contains only three main sections, because only three apply:

- “New Features of the Movie Toolbox.” This section gives background information about the new or changed features in the Movie Toolbox since the 1.5 software release, which was documented in *Inside Macintosh: QuickTime*,
- “Using the Movie Toolbox.” This section describes how to use the new features of the Movie Toolbox.
- “Movie Toolbox Reference.” This section describes all the constants, data structures, and functions in the Movie Toolbox. Each function description also follows a standard format, which gives the function declaration and description of every parameter.

Format of a New Chapter

This book also contains chapters that describe completely new areas of functionality that were not documented in the *Inside Macintosh* books. These chapters follow, with a few exceptions, a standard structure. Each begins with an overview, then some chapters give programming examples, then all the constants, data types, and functions are described in a reference section. For example, Chapter 22, “QuickTime Video Effects,” contains, among others, the following sections:

- “Introduction to QuickTime Video Effects” gives you a general overview of QuickTime video effects and the features it provides.
- “Creating New Video Effects” describes some of the tasks you can accomplish using QuickTime video effects, and gives you a set of programming examples to aid in your development efforts.
- “QuickTime Video Effects Reference” describes the constants, data types, and functions for use with QuickTime video effects. Each function description also follows a standard format, which gives the function declaration and description of every parameter.

Conventions Used in This Book

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

Special Fonts

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

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

Types of Notes

There are several types of notes used in this book.

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

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

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

Development Environment

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

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

Movie Toolbox

This chapter discusses the changes to the Movie Toolbox as documented in Chapter 2 of *Inside Macintosh: QuickTime.*

New Features of the Movie Toolbox

Preloading Tracks

There are occasions when it may be useful to preload some or all of a track's media data into memory. For example, if you are developing an application that plays several movies at once, you may want to load the smaller movies into memory in order to reduce CD-ROM seek activity. Text tracks, which are typically rather small, are good candidates for preloading; in many cases you can load a movie's entire text track into memory. Another good use of preloading is to preload small music tracks that play over scene changes, giving the interactive experience a more continuous feel.

QuickTime 2.0 expanded your options for preloading tracks. In the past, applications could use the `Load...IntoRAM` functions to preload a movie, track, or media. Now you can establish preloading guidelines as part of a track's definition. The Movie Toolbox then automatically preloads the track, according to those guidelines, every time the movie is played, without any special effort by applications. You establish these preloading guidelines by calling the new `SetTrackLoadSettings` function. (See "Enhancing Movie Playback Performance" (page 91) for more information about this function). Note that the preloading information is preserved in flattened movies.

Hints

QuickTime 1.6.1, 2.0, and 3 defined several new movie and media playback hints. These new hints work with the `SetMoviePlayHints` and `SetMediaPlayHints` functions.

`hintsAllowBlacklining`
: Instructs the Movie Toolbox to use blacklining (displaying every other line of the image) when playing a movie. Blacklining increases the speed of movie playback while decreasing the image quality. This hint is ignored if the decompressor for the movie data does not support blacklining.

`hintsDontPurge`
: Instructs the toolbox not to dispose of movie data after playing it. The toolbox leaves the data in memory, in a purgeable handle. This can enhance the playback of small movies that are looping. However, it may consume large amounts of memory and therefore affect the performance of the Memory Manager. Use this hint carefully.

`hintsInactive`
: Tells the toolbox that the movie is not in an active window. This can allow the toolbox to allocate scarce system resources more efficiently. The movie controller component automatically sets this hint for all movies it manages.

`hintsHighQuality`
: Specifies that the given movie or media should render the highest quality. For example, the video media handler should turn off fast dithering and use high-quality dithering.

  Because rendering at the highest quality takes much more time and memory than rendering at a lesser quality, this mode is usually not appropriate for real-time playback. However, since this mode generates higher quality images, it is useful when recompressing.

Data References

The Movie Toolbox now fully supports a media that refers to data in more than one file. In the past, a media was restricted to a single data file. By allowing a single media to refer to more than one file, the toolbox allows better playback performance and easier editing, primarily by reducing the number of tracks in a movie. Use the new `SetMediaDefaultDataRefIndex` function (page 109) to control which of a media's files you access when you add new sample data.

Timecode Media Handler

QuickTime 2.0 introduced support for timecode tracks. Timecode tracks allow you to store external timecode information, such as SMPTE timecodes, in your QuickTime movies. QuickTime now provides a new timecode media handler that interprets the data in these tracks.

See "Timecode Media Handler Functions" (page 124) for information about these functions.

Track References

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

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

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

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

See "Working With Track References" (page 102) for detailed descriptions of the Movie Toolbox functions that allow you to work with track references.

Modifier Tracks

The addition of **modifier tracks** in QuickTime 2.1 introduced new capabilities for creating dynamic movies. (A modifier track sends data to another track; by comparision, a track reference is an association.) For example, instead of playing video in a normal way, a video track can send its image data to a sprite track. The sprite track then uses that video data to replace the image of one of its sprites. When the movie is played, the video track appears as a sprite. (For more information about sprites and sprite tracks, refer to the book "Programming With QuickTime Sprites.")

Modifier tracks are not a new type of track. Instead, they are a new way of using the data in existing tracks. A modifier track does not present its data, but sends it to another track that uses the data to modify how it presents its own data. Any track can be either a sender or a presenter, but not both. Previously, all tracks were presenters.

Another use of modifier tracks is to store a series of sound volume levels, which is what occurs when you work with a tween track. (For more information about tweens and tween tracks, see Chapter 13, "Tween Media Handler.") These sound levels can be sent to a sound track as it plays to dynamically adjust the volume. A similar use of modifier tracks is to store location and size information. This data can be sent to a video track to cause it to move and resize as it plays.

Because a modifier track can send its data to more than one track, you can easily synchronize actions between multiple tracks. For example, a single modifier track containing matrices as its samples can make two separate video tracks follow the same path.

See "Creating Movies With Modifier Tracks" beginning on page 52 for more information about using modifier tracks.

Limitations of Spatial Modifier Tracks

A modifier track may cause a track to move outside of its original boundary regions. This may present problems, since applications do not expect the dimensions or location of a QuickTime movie to change over time.

To ensure that a movie maintains a constant location and size, the Movie Toolbox limits the area in which a spatially modified track can be displayed. A movie's "natural" shape is defined by the region returned by `GetMovieBoundsRgn`. The toolbox clips all spatially modified tracks against the region returned by `GetMovieBoundsRgn`. This means that a track can move outside of its initial boundary regions, but it cannot move beyond the combined initial boundary regions of all tracks in the movie. Areas uncovered by a moving track are handled by the toolbox in the same way as areas uncovered by tracks with empty edits.

For more information about how QuickTime handles uncovered areas, see the description of the `SetMovieCoverProcs` function on page 2-156 of *Inside Macintosh: QuickTime.*

If a track has to move through a larger area than that defined by the movie's boundary region, the movie's boundary region can be enlarged to any desired size by creating a spatial track (such as a video track) of the desired size but with no data. As long as the track is enabled, it contributes to the boundary regions of the movie.

Media Handler Support

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

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

Table 1-1 Input types supported by each Apple-supplied media handler

	Video	Text	Sound	MPEG	Music	Sprite	Time-code	3D
Matrix	•	•		•		•	•	•
Graphics mode	•	•		•		•	•	•
Clip	•	•		•		•	•	•
Volume			•	•	•			

	Video	Text	Sound	MPEG	Music	Sprite	Time-code	3D
Balance			•	•	•			
Sprite image						•		•
3D sound			•		•			

Data Handler Components

QuickTime 2.0 introduced a memory-based data handler. This data handler component works with movie data that is stored in memory (referenced by a handle) instead of in a file. This data handler has a component subtype value of

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

To create a movie that uses the handle data handler, set the data reference type to `HandleDataHandlerSubType` when you call the `NewTrackMedia` function. Note that the movie data in memory is not automatically saved with the movie. If you want to save the data that is in memory, use the `FlattenMovie` or `InsertTrackSegment` functions to copy the data from memory to a file. Note that there is a special flag for `FlattenMovie` and data handlers. For more information, see “Flattening Flags” beginning on page 72.

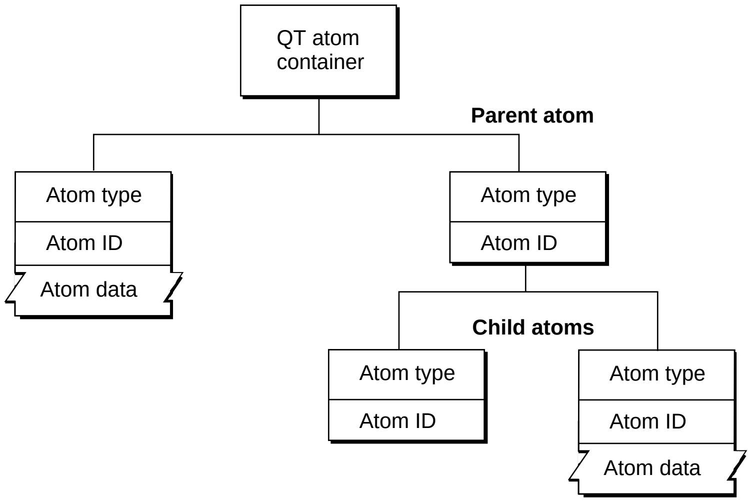
The handle data handler does not use aliases as its data reference, and therefore does not use alias handles. Rather, it uses 4-byte memory handles as its data reference. The data reference contains the actual handle that stores the needed data. If you pass a handle value of `nil`, the data handler allocates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

Although data handler components have existed since QuickTime 1.0, their interface was publicly defined for the first time in QuickTime 2.0. If you are interested in developing a data handler, refer to Chapter 1, “Data Handler Components.”

QuickTime Atoms

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

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



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

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

- by its parent atom, type, and ID

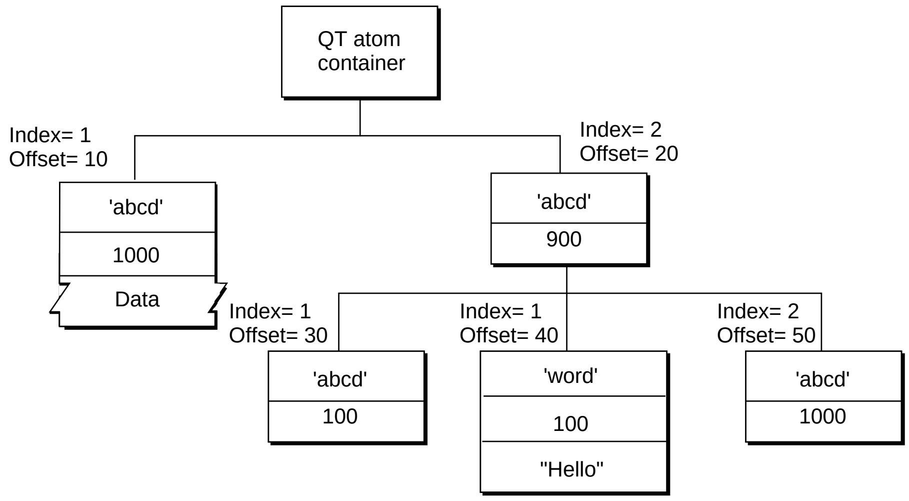
You can store and retrieve atoms in a QT atom container by index, ID, or both. For example, to use a QT atom container as a dynamic array or tree structure, you can store and retrieve atoms by index. To use a QT atom container as a database, you can store and retrieve atoms by ID. You can also create,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

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

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

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

Figure 1-2 QT atom container example



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

For more information about the internal structure of QT atoms and atom containers, see the book *QuickTime File Format Specification,* May 1996.

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

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

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

Support for Access Keys

Certain compression formats, such as the Intel Indeo format for video compression, allow data to be encrypted when it is compressed. For software to gain access to the encrypted data, the access key for the data must be registered with QuickTime. For example, a CD-ROM title would register one or more access keys for encrypted video data it presents. Once the access keys are registered, the keys are available to the decompressor component that will be used to decompress the data. The CD-ROM title can then use the encrypted data in the same way it uses unencrypted data.

QuickTime 3 includes functions that allow software to register, unregister, and get access keys. In addition, the QuickTime settings control panel lets users enter, delete, and modify access keys.

Although most access keys are character strings, an access key can be of any data type.

Scope of Access Keys

The scope of an access key is determined when it is registered. If a key is registered as a system access key, data protected by that key is available to any QuickTime client on the computer. If a key is registered as an application access key, data protected by the key is available only to QuickTime clients of the application that registers the key. For example, if a single access key is used to encrypt all the data on a CD-ROM, and the CD-ROM title registers the key as an application access key, the data on the CD-ROM is available only to the title and inaccessible to all other applications. This prevents users from browsing through data on the CD-ROM that is meant to remain hidden.

Access keys entered in the QuickTime control panel are always registered as system access keys.

Access Key Types

Access keys are grouped by type. For example, there can be an access key type specifically for the Intel Indeo video decompressor. Grouping access keys lets a decompressor request only those keys that apply to it. This speeds operations involving large numbers of keys, which might otherwise interfere with the performance of real-time operations. The functions for using access keys all require an access type.

Compressed Movie Resources

Your software can read movie files that contain compressed movie resources. Compressing a movie resource can reduce the size of the resource by as much as 50%.

Applications or other software do not have to be modified to use compressed movie resources. They must, however, use QuickTime 3 or later; earlier versions return an error when attempting to open movie files that contain compressed movie resources.

Compressing Movie Resources When Flattening

An application can create a compressed movie resource when flattening a movie by calling the `FlattenMovieData` or `FlattenMovie` function with the `flattenCompressMovieResource` flag set. This compresses the movie resource stored in the file's data fork, but not the movie resource stored in the resource fork on Mac OS systems. Compressing just the data fork is sufficient for most purposes, including transmission of movies over the Internet, where only the data fork is used.

Support for Progressive Downloads

The QuickTime 3 Movie Toolbox includes support for progressive downloads, which allow part of a movie to be displayed before all of its data has been received over a network or other slow link.

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

For more information, see "Functions for Progressive Downloads" beginning on page 162.

Using the Movie Toolbox

This section describes new or changed operations your application can perform with the Movie Toolbox.

Loading a Movie

QuickTime 2.0 made the data handler component interface available to developers. Data handlers provide a way to access data stored in any location, in any kind of container. Using a data handler, you can access data on a Macintosh hard disk, stored in memory, or stored on a network volume on a non-HFS volume. Although data handlers allow movie data to be stored on any kind of device, before QuickTime 2.0 the movie resource had to be stored on an HFS volume. QuickTime 2.1 provided you with a new function, named `NewMovieFromDataRef`, that allows a movie to be created from any device that has a corresponding data handler. Use the `NewMovieFromDataRef` function (page 86) when you need to instantiate movies from other types of devices.

Creating Movies With Modifier Tracks

QuickTime 2.1 added additional functionality for media handlers. By way of modifier tracks, a media handler can now send its data to another media handler rather than presenting its media directly. See "Modifier Tracks" (page 44) for a complete discussion of this feature.

To create a movie with modifier tracks, first you create a movie with all the desired tracks, then you create the modifier track. To link the modifier track to the track that it modifies, use the `AddTrackReference` function as shown in Listing 1-1.

Listing 1-1 Linking a modifier track to the track it modifies

```
long addedIndex;

AddTrackReference(aVideoTrack, aModifierTrack, kTrackModifierReference,
      &addedIndex);
```

The reference doesn't completely describe the modifier track's relationship to the track it modifies. Instead, the reference simply tells the modifier track to send its data to the specified track. The receiving track doesn't know what it should do with that data. A single track may also be receiving data from more than one modifier track.

To describe how each modifier input should be used, each track's media also has an **input map.** The media's input map describes how the data being sent to each input of a track should be interpreted by the receiving track. After creating the reference, it is necessary to update the receiving track's media input map. When `AddTrackReference` is called, it returns the index of the reference added. That index is the index of the input that needs to be described in the media input map. If the modifier track created above contains regions to change the shape of the video track, the code shown in Listing 1-2 updates the input map appropriately.

Listing 1-2 Updating the input map

```
QTAtomContainer inputMap;
QTAtom inputAtom;
OSType inputType;

Media aVideoMedia = GetTrackMedia(aVideoTrack);
GetMediaInputMap (aVideoMedia, &inputMap);

QTInsertChild(inputMap, kParentAtomIsContainer, kTrackModifierInput,
        addedIndex, 0,0, nil, &inputAtom);

inputType = kTrackModifierTypeClip;
QTInsertChild (inputMap, inputAtom, kTrackModifierType, 1, 0,
        sizeof(inputType), &inputType, nil);
```

```
SetMediaInputMap(aVideoMedia, inputMap);
QTDisposeAtomContainer(inputMap);
```

The media input map allows you to store additional information for each input. In the preceding example, only the type of the input is specified. In other types of references, you may need to specify additional data.

When a modifier track is playing an empty track edit, or is disabled or deleted, all receiving tracks are notified that the track input is inactive. When an input becomes inactive, it is reset to its default value. For example, if a track is receiving data from a clip modifier track and that input becomes inactive, the shape of the track reverts to the shape it would have if there were no clip modifier track.

Creating and Disposing of Atom Containers

Before you can add atoms to an atom container, you must first create the container by calling `QTNewAtomContainer`. The code sample shown in Listing 1-3 calls `QTNewAtomContainer` to create an atom container.

Listing 1-3 Creating a new atom container

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

When you have finished using an atom container, you should dispose of it by calling the `QTDisposeAtomContainer` function. The sample code shown in Listing 1-4 calls `QTDisposeAtomContainer` to dispose of the `spriteData` atom container.

Listing 1-4 Disposing of an atom container

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

Creating New Atoms

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

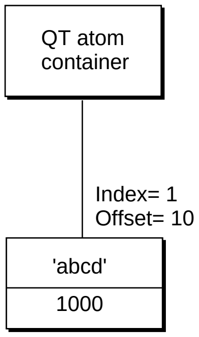
`QTInsertChild` inserts the atom in the parent's child list at the index specified by the `index` parameter; any existing atoms at the same index or greater are moved toward the end of the child list. If you specify a value of 0 for the `index` parameter, `QTInsertChild` inserts the atom at the end of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

The code sample in Listing 1-5 creates a new QT atom container and calls `QTInsertChild` to add an atom. The resulting QT atom container is shown in Figure 1-3. The offset value 10 is returned in the `firstAtom` parameter.

Listing 1-5 Creating a new QT atom container and calling `QTInsertChild` to add an atom.

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

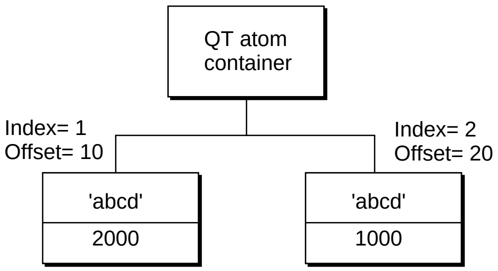
Figure 1-3 QT atom container after inserting an atom



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

```
QTAtom secondAtom;

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

Figure 1-4 QT atom container after inserting a second atom



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

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

Listing 1-6 shows how the `QTInsertChild` function inserts a leaf atom into the atom container `sprite`. The new leaf atom contains a sprite image index as its data.

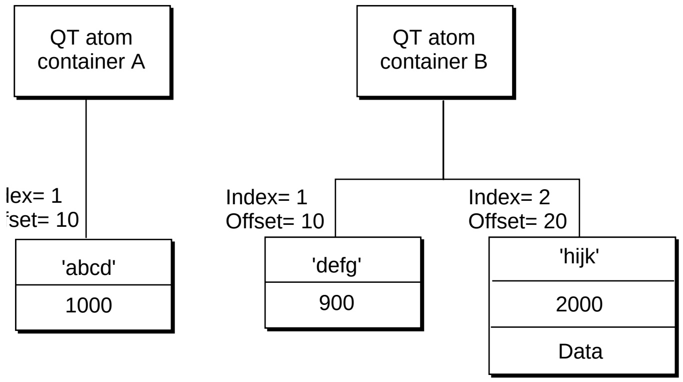
Listing 1-6 Inserting a child atom

```
if ((propertyAtom = QTFindChildByIndex (sprite, kParentAtomIsContainer,
    kSpritePropertyImageIndex, 1, nil)) == 0)

    FailOSErr (QTInsertChild (sprite, kParentAtomIsContainer,
        kSpritePropertyImageIndex, 1, 1, sizeof(short),&imageIndex,
        nil));
```

Copying Existing Atoms

QuickTime provides several functions for copying existing atoms within an atom container. The `QTInsertChildren` function inserts a container of atoms as children of a parent atom in another atom container. Figure 1-5 shows two example QT atom containers, A and B.

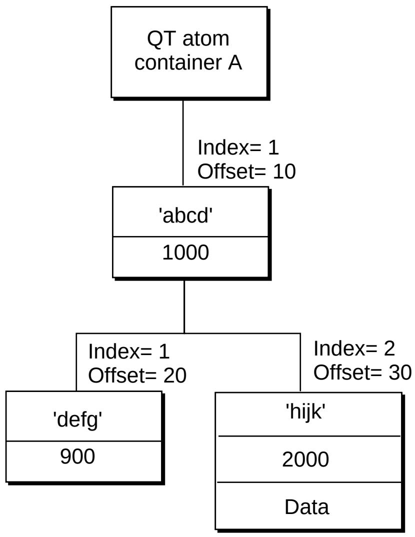
Figure 1-5 Two QT atom containers, A and B



The following code sample calls `QTFindChildByID` to retrieve the offset of the atom in container A. Then, the code sample calls the `QTInsertChildren` function to insert the atoms in container B as children of the atom in container A. Figure 1-6 shows what container A looks like after the atoms from container B have been inserted.

```
QTAtom targetAtom;

targetAtom = QTFindChildByID (containerA, kParentAtomIsContainer, 'abcd',
    1000, nil);

FailOSErr (QTInsertChildren (containerA, targetAtom, containerB));
```

Figure 1-6 QT atom container after child atoms have been inserted



In Listing 1-7, the `QTInsertChild` function inserts a parent atom into the atom container `theSample`. Then, the code calls `QTInsertChildren` to insert the container `theSprite` into the container `theSample`. The parent atom is `newSpriteAtom`.

Listing 1-7 Inserting a container into another container

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

FailOSErr (QTInsertChildren (theSample, newSpriteAtom, theSprite));
```

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

Retrieving Atoms From an Atom Container

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

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

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

Listing 1-8 shows the sample code function `SetSpriteData`, which updates an atom container that describes a sprite. (For more information about sprites and the Sprite Toolbox, refer to the book "Programming with QuickTime Sprites.") For each property of the sprite that needs to be updated, `SetSpriteData` calls `QTFindChildByIndex` to retrieve the appropriate atom from the atom container. If the atom is found, `SetSpriteData` calls `QTSetAtomData` to replace the atom's data with the new value of the property. If the atom is not found, `SetSpriteData` calls `QTInsertChild` to add a new atom for the property.

Listing 1-8 Finding a child atom by index

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

    // if the sprite’s visible property has a new value
    if (visible)
    {
        // retrieve the atom for the visible property --
        // if none exists, insert one
        if ((propertyAtom = QTFindChildByIndex (sprite,
            kParentAtomIsContainer, kSpritePropertyVisible, 1,
            nil)) == 0)
            FailOSErr (QTInsertChild (sprite, kParentAtomIsContainer,
                kSpritePropertyVisible, 1, 1, sizeof(short), visible,
                nil))

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

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

You can call the `QTFindChildByID` function to search for and retrieve a parent atom’s child by its type and ID. The sample code function `AddSpriteToSample`, shown in Listing 1-9, adds a sprite, represented by an atom container, to a key sample, represented by another atom container. `AddSpriteToSample` calls `QTFindChildByID` to determine whether the atom container `theSample` contains an atom of type `kSpriteAtomType` with the ID `spriteID`. If not, `AddSpriteToSample` calls `QTInsertChild` to insert an atom with that type and ID. A value of 0 is passed for the `index` parameter to indicate that the atom should be inserted at the end of the child list. A value of 0 is passed for the `dataSize` parameter to indicate that the atom does not have any data. Then, `AddSpriteToSample` calls

QTInsertChildren to insert the atoms in the container theSprite as children of the new atom. FailIf and FailOSErr are macros that exit the current function when an error occurs.

Listing 1-9 Finding a child atom by ID

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

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

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

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

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

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

Modifying Atoms

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

You use the QTSetAtomData function to update the data associated with a leaf atom in an atom container. The QTSetAtomData function replaces a leaf atom's data with new data. The code sample in Listing 1-10 calls QTFindChildByIndex to determine whether an atom container contains a sprite's visible property. If so, the sample calls QTSetAtomData to replace the atom's data with a new visible property.

Listing 1-10 Modifying an atom's data

```
QTAtom propertyAtom;

// if the atom isn't in the container, add it
if ((propertyAtom = QTFindChildByIndex (sprite, kParentAtomIsContainer,
    kSpritePropertyVisible, 1, nil)) == 0)
    FailOSErr (QTInsertChild (sprite, kParentAtomIsContainer,
        kSpritePropertyVisible, 1, 0, sizeof(short), visible, nil))

// if the atom is in the container, replace its data
else
    FailOSErr (QTSetAtomData (sprite, propertyAtom, sizeof(short),
        visible));
```

Removing Atoms From an Atom Container

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

The code sample shown in Listing 1-11 adds override samples to a sprite track to animate the sprites in the sprite track. The sample and spriteData variables are atom containers. The `spriteData` atom container contains atoms that describe a single sprite. The `sample` atom container contains atoms that describes an override sample.

Each iteration of the `for` loop calls `QTRemoveChildren` to remove all atoms from both the `sample` and the `spriteData` containers. The sample code updates the index of the image to be used for the sprite and the sprite's location and calls `SetSpriteData` (Listing 1-8), which adds the appropriate atoms to the `spriteData` atom container. Then, the sample code calls `AddSpriteToSample` (Listing 1-9) to add the `spriteData` atom container to the `sample` atom container. Finally, when all the sprites have been updated, the sample code calls `AddSpriteSampleToMedia` to add the override sample to the sprite track.

Listing 1-11 Removing atoms from a container

```
QTAtomContainer sample, spriteData;

// ...
// add the sprite key sample
// ...

// add override samples to make the sprites spin and move
for (i = 1; i <= kNumOverrideSamples; i++)
{
    QTRemoveChildren (sample, kParentAtomIsContainer);
    QTRemoveChildren (spriteData, kParentAtomIsContainer);

    // ...
    // update the sprite:
    // - update the imageIndex
    // - update the location
    // ...

    // add atoms to spriteData atom container
    SetSpriteData (spriteData, &location, nil, nil, &imageIndex);

    // add the spriteData atom container to sample
    err = AddSpriteToSample (sample, spriteData, 2);

    // ...
    // update other sprites
    // ...

    // add the sample to the media
    err = AddSpriteSampleToMedia (newMedia, sample,
        kSpriteMediaFrameDuration, false);
}
```

Using Access Keys

This section provides code examples that illustrate how to perform the following tasks:

- register an application access key
- unregister a system access key
- get access keys of a particular type
- use access keys in the QuickTime control panel

Registering an Access Key

Listing 1-12 illustrates how to register an application access key.

Listing 1-12 Registering an application access key

```
OSErr myErr = 0;
Str255 keyType = doomCDKeyType;
long flags = AccessKeySystemFlag;
handle keyHdl;

myErr = PtrToHand ((Ptr)”keykeykey”, &keyHdl, sizeof(“keykeykey”)-1);
myErr = QTRegisterAccessKey (keyType, flags, keyHdl);
```

Listing 1-13 illustrates how to register a system access key.

Listing 1-13 Registering a system access key

```
OSErr myErr = 0;
Str255 keyType = doomCDKeyType;
long flags = 0;
handle keyHdl;

myErr = PtrToHand ((Ptr)”keykeykey”, &keyHdl, sizeof(“keykeykey”)-1);
myErr = QTRegisterAccessKey (keyType, flags, keyHdl);
```

Unregistering an Access Key

Listing 1-14 illustrates how to unregister a system access key.

Listing 1-14 Unregistering an access key

```
OSErr myErr = 0;
Str255 keyType = doomCDKeyType;
long flags = AccessKeySystemFlag;
handle keyHdl;

myErr = PtrToHand ((Ptr)”keykeykey”, &keyHdl, sizeof(“keykeykey”)-1);
myErr = QTUnRegisterAccessKey (keyType, flags, keyHdl);
```

Getting Access Keys

Listing 1-15 illustrates how a decompressor can get application access keys of a particular type.

Listing 1-15 Getting access keys

```
OSErr myErr = 0;
Str255 keyType = doomCDKeyType;
long flags = 0;
handle keyHdl;

myErr = PtrToHand ((Ptr)”keykeykey”, &keyHdl, sizeof(“keykeykey”)-1);
myErr = QTGetAccessKeys (keyType, flags, keyHdl);
```

Access Keys in the QuickTime Control Panel

Users can enter, edit, and delete system access keys in the QuickTime control panel. The controls for these operations are illustrated in Figure 1-7, Figure 1-8, and Figure 1-9.

Figure 1-7 Click the Add button to add an access key

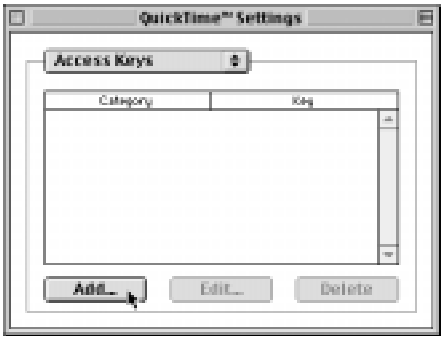
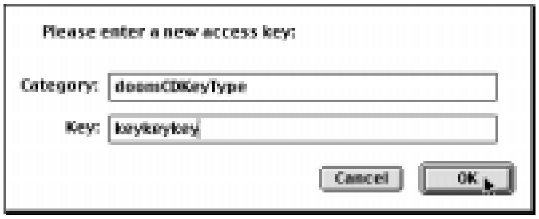


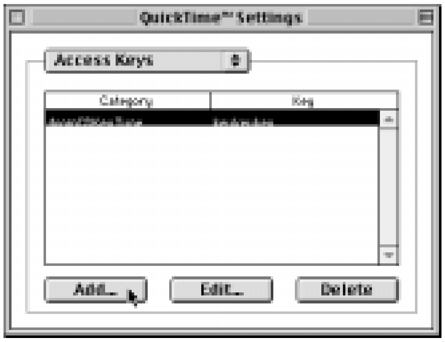
Figure 1-8 Enter the category (key type) and key, then click OK


Please enter a new access key:

Category: doomCDKeyType

Key: keykeykey

Figure 1-9 When you're done, a key can be edited or modified



Supporting Progressive Downloads

The Movie Toolbox includes functions for supporting progressive downloads of movies. These functions, which are listed in "Functions for Progressive Downloads" beginning on page 162, are illustrated in the following sections.

Displaying a Progressively Downloaded Movie

Listing 1-16 illustrates how to use the `QTMovieNeedsTimeTable` function, to find out if a movie if being progressively downloaded. and the `GetMaxLoadedTimeInMovie` function, to find out how much of the movie has been downloaded.

Listing 1-16 Displaying a progressively downloaded movie

```
WindowPtr   movieWindow;
Movie       theMovie;
Boolean     needsTimeTable;
TimeValue   loadedTime = -1;

err = GetDisplayedMovie (&movieWindow, &theMovie);
err = QTMovieNeedsTimeTable (theMovie, &needsTimeTable);

if (needsTimeTable)
{
    err = GetMaxLoadedTimeInMovie (theMovie, &loadedTime);
    // Display the movie up to the current end
}
```

Creating a Track Time Table

Listing 1-17 illustrates how to use the `MakeTrackTimeTable` function to create a time table for the tracks in a movie.

Listing 1-17 Creating a time table for movie tracks

```
OSErr       theErr = noErr;
WindowPtr   movieWindow = nil;
Movie       theMovie = nil;
Track       theTrack;
long        trackCount,
            trackIndex,
            retDataRefSkew;
TimeValue   startTime,
            endTime,
            timeIncrement;
short       firstDataRefIndex,
            lastDataRefIndex;
Handle      offsets,
            offsetsSet [32];

err = GetDisplayedMovie (&movieWindow, &theMovie);

timeIncrement = GetMovieTimeScale (theMovie);
firstDataRefIndex = -1;
lastDataRefIndex = -1;
trackCount = GetMovieTrackCount (theMovie);

for (trackIndex = 1; trackIndex <= trackCount; trackIndex++)
{
    theTrack = GetMovieIndTrack (theMovie, trackIndex);
    offsets = NewHandle (0);
    offsetsSet [trackIndex - 1] = offsets;
    startTime = GetTrackOffset (theTrack);
    endTime = GetTrackDuration (theTrack);
    err = MakeTrackTimeTable (theTrack, (long **) offsets,
                startTime, endTime, timeIncrement,
                firstDataRefIndex, lastDataRefIndex, &retDataRefSkew);
}

// dispose handles
for (trackIndex = 1; trackIndex <= trackCount; trackIndex++)
{
    DisposeHandle (offsetsSet [trackIndex - 1]);
}
```

Creating a Media Time Table

Listing 1-18 illustrates how to use the `MakeMediaTimeTable` function to create a time table for media.

Listing 1-18 Creating a time table for media

```
OSErr        theErr = noErr;
WindowPtr    movieWindow = nil;
Movie        theMovie = nil;
Track        theTrack;
Media        theMedia;
long         trackCount,
             trackIndex,
             retDataRefSkew;
TimeValue    startTime,
             endTime,
             timeIncrement;
short        firstDataRefIndex,
             lastDataRefIndex;
Handle       offsets,
             offsetsSet [32];

err = GetDisplayedMovie (&movieWindow, &theMovie);

firstDataRefIndex = -1;
lastDataRefIndex = -1;

trackCount = GetMovieTrackCount (theMovie);
```

```
for (trackIndex = 1; trackIndex <= trackCount; trackIndex++)
{
    theTrack = GetMovieIndTrack (theMovie, trackIndex);
    theMedia = GetTrackMedia (theTrack);
    offsets = NewHandle (0);
    offsetsSet [trackIndex - 1] = offsets;
    startTime = GetTrackOffset (theTrack);
    startTime = TrackTimeToMediaTime (startTime, theTrack);
    endTime = GetTrackDuration (theTrack);
    endTime = TrackTimeToMediaTime (endTime, theTrack);
    timeIncrement = GetMediaTimeScale (theMedia);
    err = MakeMediaTimeTable (theMedia, (long **) offsets,
                startTime, endTime, timeIncrement, firstDataRefIndex,
                lastDataRefIndex, &retDataRefSkew);
}

// dispose handles
for (trackIndex = 1; trackIndex <= trackCount; trackIndex++)
{
    DisposeHandle (offsetsSet [trackIndex - 1]);
}
```

Movie Toolbox Reference

This section describes the new and changed constants, data types, and functions in the Movie Toolbox.

Constants

This section describes the new constants provided by the Movie Toolbox.

Movie Exporting Flags

The `flags` parameter for the `ConvertMovieToFile` function specifies a set of movie conversion flags. QuickTime 2.0 provided these additional flag:

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

Flag descriptions

`showUserSettingsDialog`
: If this flag is set, the Save As dialog box will be displayed to allow the user to choose the type of file to export to, optional export settings, and the file name to export to.

`movieToFileOnlyExport`
: If this flag is set and the `showUserSettingsDialog` flag is set, the Save As dialog box restricts the user to those file formats that are supported by movie data export components. If this flag is not set, the user will also be able to save the movie either as a self-contained movie or as a reference movie.

`movieFileSpecValid`
: If this flag is set and the `showUserSettingsDialog` flag is set, the `name` field of the `outputFile` parameter is used as the default name of the exported file in the Save As dialog box.

Movie Importing Flags

The `flags` parameter for the `ConvertFileToMovieFile` and `PasteHandleIntoMovie` functions specifiy a set of movie conversion flags. QuickTime 1.6.1 provided one additional flag:

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

Flag description

`showUserSettingsDialog`
: If this flag is set, the user settings dialog box for that import operation can be displayed. For example, when importing a picture, this flag would display the standard compression

dialog box so that the user could select a compression method.

Flattening Flags

The flags parameter for the FlattenMovieData function specifies a set of movie flattening flags. QuickTime 2.1 provided one new flag that you must set when specifying a data reference to flatten a movie to, instead of a file:

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

Flag description

flattenFSSpecPtrIsDataRefRecordPtr
Set this flag to 1 if the FSSpec pointer is a DataReferencePtr. This capability enables you to flatten movies for devices other than file systems.

Interesting Times Flags

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

These functions work well for most media types, including video and text. However, because QuickTime stores an entire MPEG stream as a single sample, stepping to the next sample skips to the end of the sequence. To solve this problem, QuickTime 2.1 introduced a new flag for the interesting time calls: nextTimeStep. This flag returns the time of the next frame, even if there are multiple frames per sample, for all media types including video, text, and MPEG. Applications that implement single stepping capabilities should always use this flag instead of nextTimeMediaSample.

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

Flag description

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

Full Screen Flags

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

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

Flag description

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

Text Sample Display Flags

The `displayflags` parameter for the `AddTESample` and `AddTextSample` functions control the behavior of the text media handler. QuickTime 2.5 provides these additional flags:

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

Flag description

`dfContinuousScroll`
: If this flag is set, the text media handler lets new samples cause previous samples to scroll out. You must also set `dfScrollIn` or `dfScrollOff`, or both, for this to take effect.

`dfFlowHoriz`
: If this flag is set, the text media handler lets horizontally scrolled text flow within the text box instead of extending to the right.

`dfContinuousKaraoke`
: If this flag is set, the text media handler ignores the starting offset when highlighting text. Instead, it highlights text from the beginning of the text sample to the ending offset.

`dfDropShadow`
: If this flag is set, the text media handler displays text with a drop shadow. If you use the `TextMediaSetTextSampleData` function, the position and translucency of the drop shadow is under your application's control. For more information, see "TextMediaSetTextSampleData" (page 140).

`dfAntiAlias`
: If this flag is set, the text media handler displays text with anti-aliasing. Note that although anti-aliased text looks smoother, anti-aliasing can slow down performance.

`dfKeyedText`	If this flag is set, the text media handler renders text over the background without drawing the background color. This technique is also known as "masked text."
`dfInverseHilite`	If this flag is set, the text media handler highlights text using inverse video instead of the highlight color.
`dfTextColorHilite`	If this flag is set, the text media handler highlights text by changing the color of the text.

Modifier Input Types

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

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

The following input types are currently defined:

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

Constant descriptions

`kTrackModifierTypeMatrix`
: Data sent to this input should be in the form of a QuickTime `MatrixRecord`. The matrix is concatenated with the track and movie matrices to determine the tracks final location and size. The matrix modifier describes relative, not absolute, position and scaling.

`kTrackModifierTypeClip`
: Data sent to this input should be in the form of a QuickDraw region. The region is intersected with the track's source box. See *Inside Macintosh: QuickTime* for details on how a movie's tracks are assembled.

`kTrackModifierTypeGraphicsMode`
: Data sent to this input should be in the form of a `ModifierTrackGraphicsModeRecord` data type. The contents of the record are used as the graphics mode setting for the track. The graphics mode is not combined with the track's current graphics mode, but rather overrides it. See "Data Types" (page 79) for information about the `ModifierTrackGraphicsModeRecord` data structure.

`kTrackModifierTypeVolume`
: Data sent to this input should be in the form of a 16-bit fixed-point number. This is the same format in which QuickTime sound volume levels are stored. The volume level is used as a scaling factor on the sound track's level. It is multiplied with the track and movie volumes to determine the track's overall volume.

`kTrackModifierTypeBalance`
: Data sent to this input should be in the form of a 8-bit fixed-point number. This is the same format in which QuickTime balance values are stored. The balance value is used as the balance setting for the track. Unlike the volume modifier, it is not concatenated with the track's current balance level, but overrides the current balance level.

`kTrackModifierTypeImage`
: Data sent to this input should be compressed video data, typically from a video track. This input type can be used with sprite and 3D tracks. For sprite tracks, the image data is used to replace the image of a specified image index in the sprite track. The index of the image to replace must be

specified in the media input map when the reference is created. For 3D tracks, the image is used to provide a texture for the object specified in the input map.

`kTrackModifierObjectMatrix`
Data sent to this input should be in the form of a QuickTime `MatrixRecord`. The matrix is sent to a particular object within the receiving track, as specified by the `kTrackModifierObjectID` atom in the input map. The matrix acts as an override to the object's current matrix. For example, the matrix could be sent to a sprite within a sprite track. It would cause the sprite to move, not the entire sprite track as would `kTrackModifierMatrix`.

`kTrackModifierObjectGraphicsMode`
Data sent to this input should be in the form of a `ModifierTrackGraphicsModeRecord` data type. The contents of the record are used to vary the opacity of an object within the track. For example, you would use data sent to this input to vary the opacity of a sprite within a sprite track, rather than modifying the opacity of the entire sprite track. See "Data Types" (page 79) for information about the `ModifierTrackGraphicsModeRecord` data structure.

`kTrackModifierType3d4x4Matrix`
Data sent to this input should be in the form of a QuickDraw 3D 4x4 matrix—`TQ3Matrix4x4`. This data is sent to objects within a QuickDraw 3D track. This matrix is concatenated with the all other matrices that effect the specified object. The structure is defined in the book *3D Graphics Programming with QuickDraw 3D 1.5.4.*

`kTrackModifierCameraData`
Data sent to this input should be in the form of a QuickDraw 3D Camera data structure—`TQ3CameraData`. This data is sent to a camera within a QuickDraw 3D track. The structure is defined in the book *3D Graphics Programming with QuickDraw 3D 1.5.4.*

`kTrackModifierSoundLocalization`
Data sent to this input should be in the form of a sound localization data record—`SSpLocalizationData`. This data overrides the sound localization settings already in use by the track.

Text Atom Types

The `dataType` parameter for the `AddTESample` and `AddTextSample` functions indicates the type of data in the handle. The following two types have been added:

```
enum {
    dropShadowOffsetType            = 'drpo',
    dropShadowTranslucencyType      = 'drpt'
};
```

Constant descriptions

`dropShadowOffsetType`
The drop shadow offset.

`dropShadowTranslucencyType`
The drop shadow translucency.

Constants for QT Atom Functions

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

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

Constants for Access Keys

This section provides details about Access Key Manager constants.

```
#define AccessKeyAtomType ‘acky'
```

This constant specifies the type of atoms in a QT atom container that contain access keys.

```
enum
{
    AccessKeySystemFlag     = 1,
};
```

This constant specifies an operation to be performed on a system access key rather than an application access key.

Data Types

This section describes new data structures provided by the Movie Toolbox.

Data Reference

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

```
struct DataReferenceRecord {
    OSType  dataRefType;
    Handle  dataRef;
};

typedef struct DataReferenceRecord DataReferenceRecord;
typedef DataReferenceRecord *DataReferencePtr;
```

Field descriptions

`dataRefType`	Specifies the type of data reference. For an alias data reference, you set the parameter to `rAliasType`, indicating that the reference is an alias. For a handle data reference, set the parameter to `HandleDataHandlerSubType`.
`dataRef`	Specifies the actual data reference. This parameter contains a handle to the information that identifies the file to be used.

The type of information stored in the handle depends on the value of the `dataRefType` parameter. For example, if your application is loading the movie from a file, this parameter would contain an alias to the movie file.

Sample Reference

The `SampleReferenceRecord` structure is used to describe information about a sample or group of similar samples. This data structure is used by the `GetMediaSampleReferences` and `AddMediaSampleReferences` functions.

```
struct SampleReferenceRecord {
    long        dataOffset;
    long        dataSize;
    TimeValue   durationPerSample;
    long        numberOfSamples;
    short       sampleFlags;
};

typedef struct SampleReferenceRecord SampleReferenceRecord;
typedef SampleReferenceRecord *SampleReferencePtr;
```

Field descriptions

`dataOffset`	Specifies the offset into the movie data file. This field specifies the offset into the file of the sample data.
`dataSize`	Specifies the total number of bytes of sample data identified by the reference. All samples referenced by a single `SampleReferenceRecord` must be the same size.
`durationPerSample`	Specifies the duration of each sample in the reference. You must specify this parameter in the media's time scale. All samples referenced by a single `SampleReferenceRecord` must be the same duration.
`numberOfSamples`	Specifies the number of samples contained in the reference.
`sampleFlag`	Contains flags that control the operation. The following flag is available (set unused flags to 0):

`mediaSampleNotSync`	Indicates whether the sample to be added is not a synchronous sample. Set this flag to 1 if the sample is not a synchronous sample. Set this flag to 0 if the sample is a synchronous sample.

Modifier Track Graphics Mode

The modifier track graphics mode structure contains information that defines the graphics mode setting for a track. Data in this structure indicates the graphics mode setting and the RGB opcolor that is used with certain graphics modes. Data sent to the `kTrackModifierTypeGraphicsMode` input type should be in the form of a modifier track graphics mode structure.

```
struct ModifierTrackGraphicsModeRecord {
    long                    graphicsMode;
    RGBColor                opColor;
};
typedef struct ModifierTrackGraphicsModeRecord
ModifierTrackGraphicsModeRecord;
```

Field descriptions

`graphicsMode`	Specifies the graphics mode setting.
`opColor`	Contains an RGB color structure indicating the opcolor to use with the graphics mode.

QT Atom

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

```
typedef long QTAtom;
```

QT Atom Type and ID

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

```
typedef long QTAtomType;
```

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

```
typedef long QTAtomID;
```

QT Atom Container

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

```
typedef Handle QTAtomContainer;
```

Functions for Getting and Playing Movies

The Movie Toolbox contains new and changed functions for getting and playing movies.

Movie Functions

NewMovieFromUserProc

The `NewMovieFromUserProc` function creates a movie from data that you provide. Your application defines a function that delivers the movie data to the Movie Toolbox. The toolbox calls your function, specifying the amount of data required and the location for the data.

```
pascal OSErr NewMovieFromUserProc (Movie *m, short flags,
                     Boolean *dataRefWasChanged, GetMovieUPP getProc,
                     void *refCon, Handle defaultDataRef,
                     OSType dataRefType);
```

`m` 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`.

`newMovieFlags`

Controls the operation of the `NewMovieFromUserProc` function. The following flags are valid (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.

`newMovieDontResolveDataRefs`

Controls how completely the 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 toolbox only looks in the specified data reference.

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

`newMovieDontAskUnresolvedDataRefs`

Controls whether the toolbox asks the user to locate files. If you set this flag to 0, the toolbox asks the user to locate files that it cannot find on available volumes. If the 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 toolbox automatically selects enabled tracks from alternate track groups. If you set this flag to 1, the toolbox does not automatically select tracks for the movie—you must enable and disable tracks yourself.

`dataRefWasChanged`
Contains a pointer to a Boolean value. The 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 to preserve these changes.

Set the `dataRefWasChanged` parameter to `nil` if you do not want to receive this information.

`getProc` Contains a pointer to a function in your application. This function is responsible for providing the movie data to the toolbox.

`refCon` Contains a reference constant (defined as a `void` pointer). The toolbox provides this value to the function identified by the `getProc` parameter.

`defaultDataRef`
Specifies the default data reference. This parameter contains a handle to the information that identifies the file to be used to resolve any data references and as a starting point for any Alias Manager searches.

The type of information stored in the handle depends upon the value of the `dataRefType` parameter. For example, if your application is loading the movie from a file, you would refer to the file's alias in the `defaultDataRef` parameter, and set the `dataRefType` parameter to `rAliasType`.

If you do not want to identify a default data reference, set the parameter to `nil`.

`dataRefType` Specifies the type of data reference. If the data reference is an alias, you must set the parameter to `rAliasType`, indicating that the reference is an alias.

DESCRIPTION

Your application must define a function that provides the movie data to the Movie Toolbox. You specify that function to the toolbox with the `getProc` parameter. That function must support the following interface:

```
pascal OSErr MyGetMovieProc (long offset, long size, void *dataPtr,
                         void *refCon);
```

`offset`	Specifies the offset into the movie resource (not the movie file). This is the location from which your function retrieves the movie data.
`size`	Specifies the amount of data requested by the toolbox, in bytes.
`dataPtr`	Specifies the destination for the movie data.
`refCon`	Contains a reference constant (defined as a `void` pointer). This is the same value you provided to the toolbox when you called the `NewMovieFromUserProc` function.

Normally, when a movie is loaded from a file (for example, by means of the `NewMovieFromFile` function), the toolbox uses that file as the default data reference. Since the `NewMovieFromUserProc` function does not require a file specification, your application should specify the file to be used as the default data reference using the `defaultDataRef` and `dataRefType` parameters.

SPECIAL CONSIDERATIONS

The 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.

RESULT CODES

`paramErr`	–50	Invalid parameter specified
`noMovieFound`	–2048	Toolbox cannot find a movie in the movie file

Memory Manager errors
Resource Manager errors

NewMovieFromFile

The `NewMovieFromFile` function (QuickTime 2.0 or later) works with some files that do not contain movie resources. In some cases, the data in a file is already sufficiently well formatted for QuickTime or its components to understand. For example, the AIFF movie data import component can understand AIFF sound files and import the sound data into a QuickTime movie.

When the `NewMovieFromFile` function encounters a file that does not contain a movie resource, the function tries to find a movie import component that can understand the data and create a movie. For more information about new capabilities of movie data import components, see Chapter 11, "Movie Data Exchange Components." This also works for MPEG, uLaw (`.AU`), and Wave (`.WAV`) file types.

NewMovieFromDataRef

Use the new `NewMovieFromDataRef` function to create a movie from any device with a corresponding data handler. You are no longer restricted to instantiating a movie from a file stored on an HFS volume. With this new function, you can now instantiate a movie from any device.

```
pascal OSErr NewMovieFromDataRef (
                        Movie *m,
                        short flags,
                        short *id,
                        Handle dataRef,
                        OSType dataRefType);
```

`m`	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`.
`flags`	Controls the operation of the `NewMovieFromDataRef` function. The following flags are valid (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.

`newMovieDontResolveDataRefs`
: Controls how completely the 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 toolbox only looks in the specified data reference.

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

`newMovieDontAskUnresolvedDataRefs`
Controls whether the toolbox asks the user to locate files. If you set this flag to 0, the toolbox asks the user to locate files that it cannot find on available volumes. If the 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 toolbox automatically selects enabled tracks from alternate track groups. If you set this flag to 1, the toolbox does not automatically select tracks for the movie—you must enable and disable tracks yourself.

`id` Contains a pointer to the field that specifies the resource containing the movie data that is to be loaded. If the field referred to by the `id` parameter is set to 0, the 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 `id` parameter. The following enumerated constant is available:

`movieInDataForkResID`
Indicates the movie was loaded from the data fork. If the resource was stored in the file's data fork, the 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 `id` parameter is set to `nil`, the toolbox loads the first movie resource it finds in the specified file and does not return that resource's ID number.
`dataRef`	Specifies the default data reference. This parameter contains a handle to the information that identifies the file to be used to resolve any data references and as a starting point for any Alias Manager searches. The type of information stored in the handle depends upon the value of the `dataRefType` parameter. For example, if your application is loading the movie from a file, you would refer to the file's alias in the `DataRef` parameter and set the `dataRefType` parameter to `rAliasType`. If you do not want to identify a default data reference, set the parameter to `nil`.
`dataRefType`	Specifies the type of data reference. If the data reference is an alias, you must set the parameter to `rAliasType`, indicating that the reference is an alias.

DISCUSSION

`NewMovieFromDataRef` is intended for use by specialized applications that need to instantiate movies from devices not visible to the file system. Most applications should continue to use `NewMovieFromFile`.

RESULT 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 error

ConvertFileToMovieFile

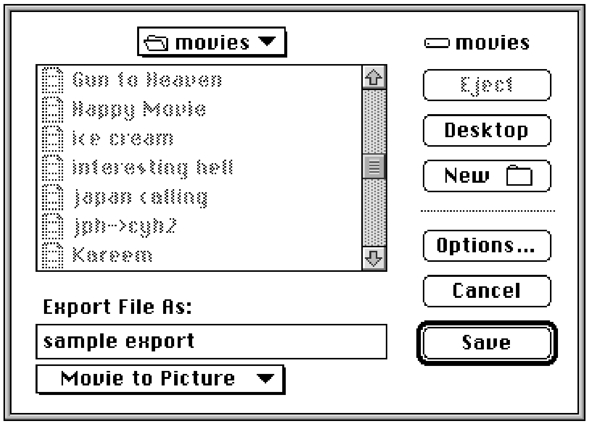
As of QuickTime 1.6.1, the `ConvertFileToMovieFile` function supports a user settings dialog box for import operations. Your application controls whether this dialog box appears by setting the value of the `flags` parameter in the `ConvertFileToMovieFile` function. This function supports the following new flag:

`showUserSettingsDialog`
Controls whether the user settings dialog box for the specified import operation can appear. Set this flag to 1 to display the user settings dialog box.

ConvertMovieToFile

The `ConvertMovieToFile` function now supports a "Save As..." dialog box. The dialog box allows the user to specify the file name and type. Supported types include standard QuickTime movies, flattened movies, single-fork flattened movies, and any format that is supported by a movie data export component. Figure 1-10 shows a sample "Save As..." dialog box.

Figure 1-10 Sample "Save As..." dialog box



Export File As:
sample export

Your application controls whether this dialog box appears by setting the value of the `flags` parameter for the `ConvertMovieToFile` function. The function supports the following flags:

`showUserSettingsDialog`
: If this bit is set, the Save As dialog box will be displayed to allow the user to choose the type of file to export to, as well as the file name to export to.

`movieToFileOnlyExport`
: If this bit is set and the `showUserSettingsDialog` bit is set, the Save As dialog box restricts the user to those file formats that are supported by movie data export components.

`movieFileSpecValid`
: If this bit is set and the `showUserSettingsDialog` bit is set, the `name` field of the `outputFile` parameter is used as the default name of the exported file in the Save As dialog box.

The following code shows how to call this function to provide a simple export capability.

```
err = ConvertMovieToFile (theMovie,     /* identifies movie */
                    nil,                /* all tracks */
                    nil,                /* no output file */
                    0,                  /* no file type */
                    0,                  /* no creator */
                    -1,                 /* script */
                    nil,                /* no resource ID */
                    createMovieFileDeleteCurFile |
                        showUserSettingsDialog |
                        movieToFileOnlyExport,
                    0);                 /* no specific component */
```

FlattenMovie and FlattenMovieData

Through the new `SetTrackLoadSettings` function, the Movie Toolbox, now allows you to set a movie's preloading guidelines when you create the movie. The preload information is preserved when you save or flatten the movie (using either the `FlattenMovie` or `FlattenMovieData` functions). In flattened movies, the tracks that are to be preloaded are stored at the start of the movie, rather than being interleaved with the rest of the movie data. This greatly improves preload performance because it is not necessary for the device storing the movie data to seek during retrieval of the data to be preloaded.

For more information about preloading, see the discussion of the `SetTrackLoadSettings` function in "Enhancing Movie Playback Performance".

Instead of flattening to a file, you can now also specify a data reference to flatten a movie to. The `FSSpec` parameter for the `FlattenMovieData` function usually contains a pointer to the file system specification for the movie file to be created. In place of the `FSSpec` parameter, QuickTime lets you pass a pointer to a data reference structure. The data reference structure defines the data reference to flatten the movie data to. For more information about this structure, see "Data Types" (page 79).

Enhancing Movie Playback Performance

Two new functions allow you to get and set a portion of a preloaded track. There is also a new function for working with modifier tracks.

SetTrackLoadSettings

The `SetTrackLoadSettings` function allows you to specify a portion of a track that is to be loaded into memory whenever it is played.

```
pascal void SetTrackLoadSettings (Track theTrack, TimeValue preloadTime,
                    TimeValue preloadDuration, long preloadFlags,
                    long defaultHints);
```

`theTrack` Specifies the track for this operation. Your application obtains this track identifier from such toolbox functions as `NewMovieTrack` and `GetMovieTrack`.

`preloadTime` Specifies the starting point of the portion of the track to be preloaded. Set this parameter to –1 if you want to preload the entire track (in this case the function ignores the `preloadDuration` parameter). This parameter should be specified using the movie's time scale.

`preloadDuration`	Specifies the amount of the track to be preloaded, starting from the time specified in the `preloadTime` parameter. If you are preloading the entire track, the function ignores this parameter.
`preloadFlags`	Controls when the toolbox preloads the track. The function supports the following flag values:

`preloadAlways`	Specifies that the toolbox should always preload this track, even if the track is disabled.
`preloadOnlyIfEnabled`	Specifies that the toolbox should preload this track only when the track is enabled.

Set this parameter to 0 if you do not want to preload the track.

`defaultHints`	Specifies playback hints for the track. You may specify any of the supported hints flags. See "Hints" (page 42) for descriptions of new hint flags in QuickTime.

DESCRIPTION

The `SetTrackLoadSettings` allows you to control how the toolbox preloads the tracks in your movie. By using these settings, you make this information part of the movie, so that the preloading takes place every time the movie is opened, without an application having to call the `LoadTrackIntoRAM` function. Consequently, you should use this feature carefully, so that your movies do not consume large amounts of memory when opened.

SPECIAL CONSIDERATIONS

The toolbox transfers this preload information when you call the `CopyTrackSettings` function. In addition, the preload information is preserved when you save or flatten a movie (using either the `FlattenMovie` or `FlattenMovieData` functions). In flattened movies, the tracks that are to be preloaded are stored at the start of the movie, rather than being interleaved with the rest of the movie data. This improves preload performance.

RESULT CODES

`invalidTrack`	–2009	This track is corrupted or invalid

GetTrackLoadSettings

The `GetTrackLoadSettings` function allows you to retrieve a track's preload information.

```
pascal void GetTrackLoadSettings (Track theTrack, TimeValue *preloadTime,
                     TimeValue *preloadDuration, long *preloadFlags,
                     long *defaultHints);
```

`theTrack` Specifies the track for this operation. Your application obtains this track identifier from such toolbox functions as `NewMovieTrack` and `GetMovieTrack`.

`preloadTime` Specifies a field to receive the starting point of the portion of the track to be preloaded. The toolbox returns a value of –1 if the entire track is to be preloaded.

`preloadDuration`
Specifies a field to receive the amount of the track to be preloaded, starting from the time specified in the `preloadTime` parameter. If the entire track is to be preloaded, this value is ignored.

`preloadFlags` Specifies a field to receive the flags that control when the toolbox preloads the track. The function supports the following flag values:

`preloadAlways` Specifies that the toolbox always preloads this track.

`preloadOnlyIfEnabled`
Specifies that the toolbox preloads this track only when the track is enabled.

`defaultHints` Specifies a field to receive the playback hints for the track.

RESULT CODES

invalidTrack	–2009	This track is corrupted or invalid

GetTrackDisplayMatrix

The GetTrackDisplayMatrix function returns a matrix that is the concatenation of all matrices currently effecting the track's location, scaling, and so on. This includes the movie's matrix, the track's matrix, and the modifier matrix. Since modifier information is passed between tracks at MoviesTask time, the information returned by this call represents the matrix in effect at the last MoviesTask call.

```
pascal OSErr GetTrackDisplayMatrix(
                    Track theTrack,
                    MatrixRecord *matrix );
```

theTrack	Specifies the track for this operation. Your application obtains this track identifier from such toolbox functions as NewMovieTrack and GetMovieTrack.
matrix	Contains a pointer to a matrix structure.

Note
To determine the entire clip of a track at the current time using GetTrackDisplayBoundsRgn. The results of GetTrackDisplayBoundsRgn take into account any clip regions provided by modifier tracks. ◆

RESULT CODES

invalidTrack	–2009	This track is corrupted or invalid

Generating Pictures From Movies

When memory is low, the GetMoviePict function now reports out of memory errors instead of returning empty pictures.

Working with Progress and Cover Functions

SetMovieDrawingCompleteProc

The `SetMovieDrawingCompleteProc` function allows you to assign a drawing-complete function to a movie. The Movie Toolbox calls this function based upon guidelines you establish when you assign the function to the movie.

```
pascal void SetMovieDrawingCompleteProc (Movie theMovie, long flags,
                        MovieDrawingCompleteUPP proc, long refCon);
```

`theMovie` Specifies the movie for this operation. Your application obtains this identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle`.

`flags` Contains information that controls when your drawing complete function is called. The following values are supported:

`movieDrawingCallWhenChanged`
Specifies that the toolbox should call your drawing-complete function only when the movie has changed.

`movieDrawingCallAlways`
Specifies that the toolbox should call your drawing-complete function every time your application calls the `MoviesTask` function.

`proc` Contains a pointer to your drawing-complete function. Set this parameter to `nil` if you want to remove your function.

`refCon` Contains a value that the toolbox provides to your drawing-complete function.

DESCRIPTION

Your drawing-complete function must support the following interface:

```
typedef pascal OSErr (*MovieDrawingCompleteProcPtr)(Movie theMovie, long
                        refCon);
```

`theMovie`	Specifies the movie for this operation.
`refCon`	Contains the reference constant you supplied when your application called the `SetMovieDrawingCompleteProc` function.

IMPORTANT

Some media handlers may take less efficient playback paths when a drawing-complete function is used, so it should be used only when absolutely necessary. ▲

RESULT CODES

`invalidMovie`	–2010	This movie is corrupted or invalid

SetMovieCoverProcs

If a movie with semi-transparent tracks has a movie uncover procedure (set with the `SetMovieCoverProcs` function), the uncover procedure is now called before each frame to fill or erase the background. Before QuickTime 1.6.1, the Movie Toolbox performed the erase, which limited a cover procedure–aware application's options.

GetMovieCoverProcs

The `GetMovieCoverProcs` function allows you to retrieve the cover functions that you set with the `SetMovieCoverProcs` function.

```
pascal OSErr GetMovieCoverProcs(
                    Movie theMovie,
                    MovieRgnCoverUPP *uncoverProc,
                    MovieRgnCoverUPP *coverProc,
                    long *refcon)
```

`theMovie`	Specifies the movie for this operation. Your application obtains this identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle`.

`uncoverProc`	Where to return the current uncover procedure. This value is set to `nil` if no uncover procedure was specified.
`coverProc`	Where to return the current cover procedure. This value is set to `nil` if no cover procedure was specified.
`refcon`	Specifies a reference constant. The toolbox passes this value to your cover functions.

DISCUSSION

The `GetMovieCoverProcs` function returns the uncover and cover functions for the movie as well as the reference constant for the cover functions.

RESULT CODES

`invalidMovie`	–2010	This movie is corrupted or invalid

Functions That Modify Movie Properties

Working With Movie Spatial Characteristics

SetMovieColorTable

The `SetMovieColorTable` function allows you to associate a color table with a movie.

```
pascal OSErr SetMovieColorTable (Movie theMovie, CTabHandle ctab);
```

`theMovie`	Specifies the movie for this operation. Your application obtains this identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle`.
`ctab`	Contains a handle to the color table. Set this parameter to `nil` to remove the movie's color table.

DESCRIPTION

The color table you supply may be used to modify the palette of indexed display devices at playback time. If you are using the movie controller, be sure to set the `mcFlagsUseWindowPalette` flag. If you are not using the movie controller, you should retrieve the movie's color table (using the `GetMovieColorTable` function) and supply it to the Palette Manager.

The toolbox makes a copy of the color table, so it is your responsibility to dispose of the color table when you are done with it. If the movie already has a color table, the toolbox uses the new table to replace the old one.

The `CopyMovieSettings` function copies the movie's color table, along with the other settings information.

RESULT CODES

`invalidMovie`	–2010	The movie is corrupted or invalid

Memory Manager errors

GetMovieColorTable

The `GetMovieColorTable` function allows you to retrieve a movie's color table.

```
pascal OSErr GetMovieColorTable (Movie theMovie, CTabHandle *ctab);
```

`theMovie`	Specifies the movie for this operation. Your application obtains this identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle`.
`ctab`	Contains a pointer to a field that is to receive a handle to the movie's color table. If the movie does not have a color table, the toolbox sets the field to `nil`.

DESCRIPTION

The toolbox returns a copy of the color table, so it is your responsibility to dispose of the color table when you are done with it.

RESULT CODES

`invalidMovie`	–2010	The movie is corrupted or invalid
Memory Manager errors		

SetTrackGWorld

The `SetTrackGWorld` function allows you to force a track to draw into a particular graphics world. This graphics world may be different from that of the movie.

```
pascal void SetTrackGWorld(
                        Track theTrack,
                        CGrafPtr port,
                        GDHandle gdh,
                        TrackTransferUPP proc,
                        long refCon);
```

`theTrack`	Specifies the track for this operation. Your application obtains this identifier from such functions as `GetMovieTrack`, `GetMovieIndTrack`, and `GetMovieIndTrackType`.
`port`	Points to the graphics port structure or graphics world to which to draw the track. Set this parameter to `nil` to use the movie's 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.
`proc`	Contains a pointer to your transfer procedure. Set this parameter to `nil` if you want to remove your transfer procedure.
`refCon`	Contains a value to pass to your transfer procedure.

DISCUSSION

After the `SetTrackGWorld` function draws a track, it calls your transfer procedure to copy the track to the actual movie graphics world. When your transfer procedure is called, the current graphics world is set to the correct destination. You can also install a transfer procedure and set the graphics world to `nil`. If the graphics world is set to `nil`, `SetTrackGWorld` calls your transfer procedure only as a notification that the track has been drawn; no transfer needs to take place.

RESULT CODES

`invalidTrack`	–2009	This track is corrupted or invalid

Locating a Movie's Tracks and Media Structures

GetMovieIndTrackType

The `GetMovieIndTrackType` function allows you to search for all of a movie's tracks that share a given media type or media characteristic.

```
pascal Track GetMovieIndTrackType (Movie theMovie, long index,
                       OSType trackType, long flags);
```

`theMovie`	Specifies the movie for this operation. Your application obtains this identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle`.
`index`	Specifies the index value of the track for this operation. This is not that same as the track's index value in the movie. Rather, this parameter is an index into the set of tracks that meet your other selection criteria.
`trackType`	Contains either a media type or a media characteristic value. The toolbox applies this value to the search, and returns information about tracks that meet this criterion. You indicate whether you have specified a media type or characteristic value by setting the `flags` parameter appropriately.
`flags`	Contains flags that control the search operation. The following flags are valid (note that you may not set both `movieTrackMediaType` and `movieTrackCharacteristic` to 1):

`movieTrackMediaType`
: Indicates that the `trackType` parameter contains a media type value. Set this flag to 1 if you are supplying a media type value (such as `VideoMediaType`).

`movieTrackCharacteristic`
: Indicates that the `trackType` parameter contains a media characteristic value. Set this flag to 1 if you are supplying a media characteristic value (such as `VisualMediaCharacteristic`).

`movieTrackEnabledOnly`
: Specifies that the toolbox should only search enabled tracks. Set this track to 1 to limit the search to enabled tracks.

function result Returns the track identifier of your selected track, or `nil`.

DESCRIPTION

The toolbox returns the track identifier that corresponds to the track that meets your selection criteria. If the toolbox cannot find a matching track, in returns a value of `nil`.

Note that the `index` parameter does not work the same way that is does in the `GetMovieIndTrack` function. With the `GetMovieIndTrackType` function, the index parameter specifies an index into the set of tracks that meet your other selection criteria. For example, in order to find the third track that supports the sound characteristic, you would call the function in the following manner:

```
theTrack = GetMovieIndTrackType (theMovie,
                      3,
                      AudioMediaCharacteristic,
                      movieTrackCharacteristic);
```

RESULT CODES

`paramErr`	–50	Invalid parameter specified
`invalidMovie`	–2010	The movie is corrupted or invalid

Working With Track References

Track references allow you to relate tracks to one another. For example, this can help you identify the text track that contains the subtitles for a movie's audio track and relate that text track to a particular audio track. See "Track References" beginning on page 43 for more information about track references.

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

AddTrackReference

The `AddTrackReference` function allows you to add a new track reference to a track.

```
pascal OSErr AddTrackReference (Track theTrack, Track refTrack,
                         OSType refType, long *addedIndex);
```

`theTrack`	Identifies the track for this operation. Your application obtains this track identifier from such toolbox functions as `NewMovieTrack` and `GetMovieTrack`.
`refTrack`	Specifies the track to be identified in the track reference.
`refType`	Specifies the type of reference.
`addedIndex`	Contains a pointer to a long integer. The toolbox returns the index value assigned to the new track reference. If you do not want this information, set this parameter to `nil`.

RESULT CODES

`invalidTrack`	–2009	This track is corrupted or invalid

Memory Manager errors

DeleteTrackReference

The `DeleteTrackReference` function allows you to remove a track reference from a track.

```
pascal OSErr DeleteTrackReference (Track theTrack, OSType refType, long
                     index);
```

`theTrack`	Identifies the track for this operation. Your application obtains this track identifier from such toolbox functions as `NewMovieTrack` and `GetMovieTrack`.
`refType`	Specifies the type of reference.
`index`	Specifies the index value of the reference to be deleted. You obtain this index value when you create the track reference.

DESCRIPTION

This function deletes a track reference from a track. If there are additional track references with higher index values, the toolbox automatically renumbers those references, decrementing their index values by 1.

RESULT CODES

`paramErr`	–50	Invalid parameter specified
`invalidTrack`	–2009	This track is corrupted or invalid

Memory Manager errors

SetTrackReference

The `SetTrackReference` function allows you to modify an existing track reference. You may change the track reference so that it identifies a different track in the movie.

```
extern pascal OSErr SetTrackReference (Track theTrack, Track refTrack,
                     OSType refType, long index);
```

`theTrack`	Identifies the track for this operation. Your application obtains this track identifier from such toolbox functions as `NewMovieTrack` and `GetMovieTrack`.
`refTrack`	Specifies the track to be identified in the track reference. The toolbox uses this information to update the existing track reference.
`refType`	Specifies the type of reference.
`index`	Specifies the index value of the reference to be changed. You obtain this index value when you create the track reference.

RESULT CODES

`paramErr`	–50	Invalid parameter specified
`invalidTrack`	–2009	This track is corrupted or invalid

GetTrackReference

The `GetTrackReference` function allows you to retrieve the track identifier contained in an existing track reference.

```
pascal Track GetTrackReference (Track theTrack, OSType refType, long
                                index);
```

`theTrack`	Identifies the track for this operation. Your application obtains this track identifier from such toolbox functions as `NewMovieTrack` and `GetMovieTrack`.
`refType`	Specifies the type of reference.
`index`	Specifies the index value of the reference found. You obtain this index value when you create the track reference.
function result	Returns the track identifier contained in the specified track reference.

DESCRIPTION

This function returns the track identifier that is contained in the specified track reference. If the toolbox cannot locate the track reference corresponding to your specifications, it returns a value of `nil`.

GetNextTrackReferenceType

The `GetNextTrackReferenceType` function allows you to determine all of the track reference types that are defined for a given track.

```
pascal OSType GetNextTrackReferenceType (Track theTrack, OSType refType);
```

`theTrack`	Identifies the track for this operation. Your application obtains this track identifier from such toolbox functions as `NewMovieTrack` and `GetMovieTrack`.
`refType`	Specifies the type of reference. Set this parameter to 0 to retrieve the first track reference type. On subsequent requests, use the previous value returned by this function.
function result	Returns an operating-system data type.

DESCRIPTION

This function returns an `OSType` containing the next track reference type value defined for the track. There is no implied ordering of the returned values. When you reach the end of the track's reference types, this function sets the returned value to 0. You can use this value to stop your scanning loop.

GetTrackReferenceCount

The `GetTrackReferenceCount` function allows you to determine how many track references of a given type exist for a track.

```
pascal long GetTrackReferenceCount (Track theTrack, OSType refType);
```

`theTrack`	Identifies the track for this operation. Your application obtains this track identifier from such toolbox functions as `NewMovieTrack` and `GetMovieTrack`.
`refType`	Specifies the type of reference. The toolbox determines the number of track references of this type.
function result	Returns a long integer or 0.

DESCRIPTION

This function returns long integer that contains the number of track references of the specified type in the track. If there are no references of the type you have specified, the function returns a value of 0.

Working With Sound

The following calls operate on the static 3D sound setting for a track. By constantly setting the value it is possible for an application to make a track's sound move in 3D space. If it is necessary to store dynamically changing 3D sound settings for the track, this can be done using the modifier track mechanism in conjunction with a tween track. This process is described in the following section.

SetTrackSoundLocalizationSettings

`SetTrackSoundLocalizationSettings` replaces the current 3D sound settings for the specified track with the new `SSpLocalizationData` record contained in the settings handle. The effect of the new 3D sound setting takes place immediately. This call always stores the new record passed, even if the track or the computer is not capable of actually meeting the request. You can pass a `nil` handle to indicate that no 3D sound effects should be used for this track. When the movie is saved, the 3D sound settings is saved with it.

`SetTrackSoundLocalizationSettings` makes a copy of the handle passed, so the caller is responsible for disposing of the settings handle.

```
pascal OSErr SetTrackSoundLocalizationSettings (Track theTrack, Handle
                         settings)
```

`theTrack`	Identifies the track for this operation. Your application obtains this track identifier from such toolbox functions as `NewMovieTrack` and `GetMovieTrack`.
`settings`	The settings you want to apply, in the format of a sound sprockets `SSpLocalizationData` record.

The following example code shows how to set the static 3D sound setting for a track using `SetTrackSoundLocalizationSettings`.

```
void setTrackSoundLocalization(Track t)
{
    SSpLocalizationData loc;
    Handle h;
    OSErr err;

    loc.cpuLoad = 0;
    loc.medium = kSSpMedium_Air;
    loc.humidity = 0;
    loc.roomSize = 250;
    loc.roomReflectivity = -5;
    loc.reverbAttenuation = -5;
    loc.sourceMode = kSSpSourceMode_Localized;
    loc.referenceDistance = 1;
    loc.coneAngleCos = 0;
    loc.coneAttenuation = 0;
    loc.currentLocation.elevation = 0;
    loc.currentLocation.azimuth = 0;
    loc.currentLocation.distance = 2;
    loc.currentLocation.projectionAngle = 0;
    loc.currentLocation.sourceVelocity = 0;
    loc.currentLocation.listenerVelocity = 0;
    loc.reserved0 = 0;
    loc.reserved1 = 0;
    loc.reserved2 = 0;
    loc.reserved3 = 0;
    loc.virtualSourceCount = 0;

    err = PtrToHand(&loc, &h, sizeof(loc));
    err = SetTrackSoundLocalizationSettings(t, h);
    DisposeHandle(h);
}
```

GetTrackSoundLocalizationSettings

`GetTrackSoundLocalizationSettings` returns a handle containing a copy of the current 3D sound settings for the specified track.

```
pascal OSErr GetTrackSoundLocalizationSettings (Track theTrack, Handle
                     *settings)
```

`theTrack`	Identifies the track for this operation. Your application obtains this track identifier from such toolbox functions as `NewMovieTrack` and `GetMovieTrack`.
`settings`	The settings you want to retrieve, in the format of a sound sprockets `SSpLocalizationData` record.

DISCUSSION

If there are no 3D sound settings, the returned handle is set to `nil`. The caller of this function is responsible for disposing of the returned handle.

Functions for Editing Movies

PasteHandleIntoMovie

As of QuickTime 1.6.1, the `PasteHandleIntoMovie` function supports a user settings dialog box for import operations. Your application controls whether this dialog box appears by setting the value of the `flags` parameter in the `PasteHandleIntoMovie` function. This function supports the following new flag:

`showUserSettingsDialog`

Controls whether the user settings dialog box for the specified import operation can appear. Set this flag to 1 to display the user settings dialog box.

Adding Samples to Media Structures

SetMediaDefaultDataRefIndex

The `SetMediaDefaultDataRefIndex` function allows you to specify which of a media's data references is to be accessed during an editing session.

```
pascal OSErr SetMediaDefaultDataRefIndex (Media theMedia, short index);
```

`theMedia`	Specifies the media for this operation. Your application obtains this media identifier from such toolbox functions as `NewTrackMedia` and `GetTrackMedia`.
`index`	Specifies the data reference to access. Values of the `index` parameter range from 1 to the number of data references in the media. You can determine the number of data references by calling the `GetMediaDataRefCount` function. Once set, the default data reference index persists. Set this parameter to 0 to revert to the media's default data reference.

DESCRIPTION

Before QuickTime 2.0, the Movie Toolbox did not allow the creation of tracks that have data in several files. Therefore, there was not a mechanism for controlling which data reference is affected by a media editing session. The `SetMediaDefaultDataRefIndex` function allows you to specify the index of the data reference to be edited. After calling this function, you can start editing that data reference by calling the `BeginMediaEdits` function.

RESULT CODES

`invalidMedia`	–2008	The media is corrupted or invalid
`badDataRefIndex`	–2050	Data reference index value is invalid

SetMediaPreferredChunkSize

The `SetMediaPreferredChunkSize` function allows you to specify a maximum chunk size for a media.

```
pascal OSErr SetMediaPreferredChunkSize (Media theMedia, long
                        maxChunkSize);
```

`theMedia`	Specifies the media for this operation. Your application obtains this media identifier from such toolbox functions as `NewTrackMedia` and `GetTrackMedia`.
`maxChunkSize`	Specifies the maximum chunk size, in bytes.

DISCUSSION

The term *chunk* refers to the collection of sample data that is added to a movie when you call the `AddMediaSample` function. When QuickTime loads a movie for playback, it loads the data a chunk at a time. Consequently, both the size and number of chunks in a movie can affect playback performance. The toolbox tries to optimize playback performance by consolidating adjacent sample references into a single chunk (up to the limit you prescribe with this function).

RESULT CODES

`noMediaHandler`	–2006	Media has no media handler
`invalidMedia`	–2008	The media is corrupted or invalid

GetMediaPreferredChunkSize

The `GetMediaPreferredChunkSize` function allows you to retrieve the maximum chunk size for a media.

```
pascal OSErr GetMediaPreferredChunkSize (Media theMedia, long
                        *maxChunkSize);
```

`theMedia`	Specifies the media for this operation. Your application obtains this media identifier from such toolbox functions as `NewTrackMedia` and `GetTrackMedia`.
`maxChunkSize`	Specifies a field to receive the maximum chunk size, in bytes.

RESULT CODES

`noMediaHandler`	–2006	Media has no media handler
`invalidMedia`	–2008	The media is corrupted or invalid

Editing Tracks

The Movie Toolbox contains one new function for editing tracks.

AddEmptyTrackToMovie

The `AddEmptyTrackToMovie` function duplicates a track from a movie into the same movie or into another movie. The newly created track has the same media type and track settings as the specified track. However, no data is copied from the source track to the new track.

To copy data from the source track to the new track, use the `InsertTrackSegment` function after calling `AddEmptyTrackToMovie`.

```
pascal OSErr AddEmptyTrackToMovie(Track srcTrack,
                    Movie dstMovie,
                    Handle dataRef,
                    OSType dataRefType,
                    Track *dstTrack);
```

`srcTrack`	Specifies the source track for this operation. Your application obtains this track identifier from such toolbox functions as `NewMovieTrack` and `GetMovieTrack`.
`dstMovie`	Specifies the destination movie for this operation. This can be the same movie as the source track or a different movie.

`dataRef`	Contains a handle to the data reference. The type of information stored in the handle depends upon the data reference type specified by the `dataRefType` parameter.
`dataRefType`	Specifies the type of data reference. If the data reference is an alias, you must set the parameter to `rAliasType`, indicating that the reference is an alias.
`dstTrack`	The newly created track's identifier is returned in this parameter. If `AddEmptyTrackToMovie` fails, the resulting track identifier is set to `nil`.

DISCUSSION

The `AddEmptyTrackToMovie` function returns the newly created, empty track. This function has been available since QuickTime 2.0.

Using the Full Screen

QuickTime 2.1 introduced two functions that you can use to put a device into full-screen mode (that is, select where and when the menu bar is not visible).

BeginFullScreen

You can use the `BeginFullScreen` function to begin full-screen mode for a specified monitor.

```
pascal OSErr BeginFullScreen (
                     Ptr *restoreState,
                     GDHandle whichGD,
                     short *desiredWidth,
                     short *desiredHeight,
                     WindowPtr *newWindow,
                     RGBColor *eraseColor,
                     long flags);
```

`restoreState`	On exit, a pointer to a block of private state data that contains information on how to return from full-screen mode. This value is passed to `EndFullScreen` to enable it to return the monitor to its previous state.
`whichGD`	A handle to the graphics device to put into full-screen mode. Set this parameter to `nil` to select the main screen.
`desiredWidth`	On entry, a pointer to a short integer that contains the desired width, in pixels, of the images to be displayed. On exit, that short integer is set to the actual number of pixels that can be displayed horizontally. Set this parameter to 0 to leave the width of the display unchanged.
`desiredHeight`	On entry, a pointer to a short integer that contains the desired height, in pixels, of the images to be displayed. On exit, that short integer is set to the actual number of pixels that can be displayed vertically. Set this parameter to 0 to leave the height of the display unchanged.
`newWindow`	On entry, a window-creation value. If this parameter is `nil`, no window is created for you. If this parameter has any other value, `BeginFullScreen` creates a new window that is large enough to fill the entire screen and returns a pointer to that window in this parameter. You should not dispose of that window yourself; instead, `EndFullScreen` will do so.
`eraseColor`	The color to use when erasing the full-screen window created by `BeginFullScreen` if `newWindow` is not `nil` on entry. If this parameter is `nil`, `BeginFullScreen` uses black when initially erasing the window's content area.
`flags`	The `flags` parameter specifies a set of bit flags that control certain aspects of the full-screen mode. QuickTime defines the following constants that you can use in the `flags` parameter:

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

Flag description

`fullScreenHideCursor`
: If this flag is set, `BeginFullScreen` hides the cursor. This is useful if you are going to play a QuickTime movie and do not want the cursor to be visible over the movie.

`fullScreenAllowEvents`
: If this flag is set, your application intends to allow other applications to run (by calling `WaitNextEvent` to grant them processing time). In this case, `BeginFullScreen` does not change the monitor resolution, because other applications might depend on the current resolution.

`fullScreenDontChangeMenuBar`
: If this flag is set, `BeginFullScreen` does not hide the menu bar. This is useful if you want to change the resolution of the monitor but still need to allow the user to access the menu bar.

`fullScreenPreflightSize`
: If this flag is set, `BeginFullScreen` doesn't change any monitor settings, but returns the actual height and width that it would use if this bit were not set. This allows applications to test for the availability of a monitor setting without having to switch to it.

DISCUSSION

The `BeginFullScreen` function returns, in the `restoreState` parameter, a pointer to a block of private state information that indicates how to return from full-screen mode. You pass that pointer as a parameter to the `EndFullScreen` function.

The Macintosh human interface guidelines suggest that the menu bar must always be present, and that information must always appear in windows. However, many multimedia applications have chosen to change the look and feel of the interface based on their needs. The number of details to keep track of when doing this continues to increase. To help solve this problem, QuickTime 2.1 added functions to put a graphics device into full screen mode.

EndFullScreen

You can use the `EndFullScreen` function to end full-screen mode for a graphics device.

```
pascal OSErr EndFullScreen (Ptr fullState, long flags);
```

`fullState`	The pointer to private state information returned by a previous call to `BeginFullScreen`.
`flags`	Reserved. Set this parameter to `nil`.

DISCUSSION

The `EndFullScreen` function restores the graphics device and other settings to the state specified by the private state information pointed to by the `fullState` parameter. The resulting state is that that was in effect prior to the immediately previous call to the `BeginFullScreen` function.

Handling Update Events

QuickTime 2.1 introduced a new function, `InvalidateMovieRegion`, to use in place of the `UpdateMovie` function to indicate the area of a movie that needs to be redrawn.

InvalidateMovieRegion

Use the new `InvalidateMovieRegion` function instead of the `UpdateMovie` function to invalidate a small area of a movie. `InvalidateMovieRegion` marks all areas of the movie that intersect the `invalidRgn` parameter. The next time you call the `MoviesTask` function, the toolbox redraws the marked areas.

```
pascal OSErr InvalidateMovieRegion (
                    Movie theMovie,
                    RgnHandle invalidRgn);
```

`theMovie`	Identifies the movie whose area you wish to invalidate. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle`.
`invalidRgn`	Contains a region indicating the area of the movie to invalidate. If necessary, QuickTime will make a copy of this region. To invalidate the entire movie area, pass `nil` for this parameter.

DESCRIPTION

The `InvalidateMovieRegion` function provides a way to invalidate a portion of the movie's area instead of its entire area, as does `UpdateMovie`. This allows for higher performance update handling when a movie has many tracks or covers a large area. For handling of update events, applications should continue to use `UpdateMovie`.

RESULT CODES

`invalidMovie`	–2010	The movie is corrupted or invalid

Handling Media Sample References

You could always use `GetMediaSampleReference` to access samples in a movie one at a time. QuickTime 2.1 introduced `GetMediaSampleReferences` (note that this is the plural form of the `GetMediaSampleReference` function), which you can use to obtain information about groups of samples. QuickTime 2.1 also introduced `AddMediaSampleReferences`, which you can use to work with groups of samples that have already been added to a movie.

GetMediaSampleReferences

The `GetMediaSampleReferences` function allows your application to obtain reference information about groups of samples that are stored in a movie.

```
pascal OSErr GetMediaSampleReferences (
                         Media theMedia,
                         TimeValue time,
                         TimeValue *sampleTime,
                         SampleDescriptionHandle sampleDescriptionH,
                         long *sampleDescriptionIndex,
                         long maxNumberOfEntries,
                         long *actualNumberofEntries,
                         SampleReferencePtr sampleRefs);
```

`theMedia` Specifies the media for this operation. Your application obtains this media identifier from such toolbox functions as `NewTrackMedia` and `GetTrackMedia`. For information about these functions, see *Inside Macintosh: QuickTime.*

`time` Specifies the starting time of the sample references to be retrieved. You must specify this value in the media's time scale.

`sampleTime` Contains a pointer to a time value. The `GetMediaSampleReferences` function updates this time value to indicate the actual time of the first returned sample data. If you are not interested in this information, set this parameter to `nil`.

`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`.

`GetMediaSampleReferences` only returns a single sample description. If the sample description changes within the media, `GetMediaSampleReferences` returns only as many samples as use a single sample description. You must call it again to get the next group of samples using the next sample description.

`sampleDescriptionIndex`
Contains a pointer to a long integer. The `GetMediaSampleReferences` function returns an index value to the sample descriptions that correspond to the returned sample data. You can use this index to retrieve the media sample descriptions with the `GetMediaSampleDescription` function. If you do not want this information, set this parameter to `nil`.

`maxNumberOfEntries`
Specifies the maximum number of entries to be returned. The sample references pointer provided by the `sampleRefs` parameter

must be large enough to receive the number of entries specified by this parameter. The toolbox does not return more entries than you specify with this parameter. It may, however, return fewer.

`actualNumberOfEntries`
Contains a pointer to a long integer. The `GetMediaSampleReferences` function updates the field referred to by this parameter with the number of entries referred to by the returned reference.

`sampleRefs` Contains a pointer to the number of `SampleReferenceRecords` specified in the `maxNumberOfEntries` parameter. On return from this call, the number of sample reference records indicated by the value returned in `actualNumberOfEntries` will be filled in.

DESCRIPTION

Using this function instead of `GetMediaSampleReference` can greatly increase the performance of operations that need access to information about each sample in a movie. No information is returned from this call that wasn't previously available from `GetMediaSampleReference`.

RESULT CODES

`invalidMedia`	–2008	This media is corrupted or invalid.

Memory Manager errors

AddMediaSampleReferences

The `AddMediaSampleReferences` function allows your application to add groups of samples to a movie data file.

```
pascal OSErr AddMediaSampleReferences (
                    Media theMedia,
                    SampleDescriptionHandle sampleDescriptionH,
                    long numberOfSamples,
                    SampleReferencePtr sampleRefs,
                    TimeValue *sampleTime);
```

`theMedia`	Specifies the media for this operation. Your application obtains this media identifier from such toolbox functions as `NewTrackMedia` and `GetTrackMedia`. For information about these functions, see *Inside Macintosh: QuickTime.*
`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. A media that contains sound requires that you supply a sound description structure. For information about the Image Compression Manager and the sound description structure, see *Inside Macintosh: QuickTime.* If you do not want the sample description, set this parameter to `nil`.
`numberOfSamples`	Specifies the number of samples contained in the reference. For details, see the `AddMediaSample` function description in *Inside Macintosh: QuickTime.*
`sampleRefs`	Contains a pointer to the number of `SampleReferenceRecords` specified in the `numberOfSamples` parameter.
`sampleTime`	Contains a pointer to a time value. After adding the reference to the media, the `AddMediaSampleReferences` function returns the time where the reference was inserted, using the time scale referred to by this parameter. If you do not want to receive this information, set this parameter to `nil`.

DISCUSSION

Using this function instead of `AddMediaSampleReference` can greatly improve the performance of operations that involve adding a large number of samples to a movie at one time. `AddMediaSampleReferences` provides no capabilities that weren't previously available with `AddMediaSampleReference`.

RESULT CODES

`invalidMedia`	–2008	This media is corrupted or invalid.

Memory Manager errors

Managing the Video Frame Playback Rate

QuickTime 2.1 introduced two new functions for determining the rate at which a QuickTime movie plays back each video frame. You should use these functions for debugging.

GetVideoMediaStatistics

The `GetVideoMediaStatistics` function returns the play-back frame rate of a movie. This call can only be used on video or MPEG media handlers.

```
pascal Fixed GetVideoMediaStatistics (
                        MediaHandler mh);
```

`mh`	Contains a reference to a video media handler. You obtain this reference from the `GetMediaHandler` function.
function result	Returns the movie’s play-back frame rate in frames per second.

DESCRIPTION

The `GetVideoMediaStatistics` returns the average frame rate since the last time `ResetVideoMediaStatistics` was called. Because of sampling errors, the values returned from `GetVideoMediaStatistics` are accurate only after waiting at least one second after calling `ResetVideoMediaStatistics`.

Note
Because not all QuickTime movies have a constant frame rate, the results of this call can be difficult to interpret correctly. For this reason, the results of this function should not be displayed in a place where a novice user is likely to see it. ◆

ResetVideoMediaStatistics

Use the `ResetVideoMediaStatistics` function to reset the video media handler's counters before using `GetVideoMediaStatistics` to determine the frame rate of a movie. This call can only be used on video or MPEG media handlers.

```
pascal HandlerError ResetVideoMediaStatistics (
                       MediaHandler mh);
```

`mh`	Contains a reference to a video media handler. You obtain this reference from the `GetMediaHandler` function.
function result	Returns a handle error.

DESCRIPTION

The `ResetVideoMediaStatistics` function resets the video media handler's frame rate counters.

RESULT CODES

`badComponentInstance`	0x80008001	Invalid component instance specified.

Manipulating Media Input Maps

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

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

GetMediaInputMap

The `GetMediaInputMap` function returns a copy of the input map associated with the specified media.

```
pascal OSErr GetMediaInputMap (
                    Media theMedia,
                    QTAtomContainer *inputMap );
```

`theMedia`	Specifies the media for this operation. Your application obtains this media identifier from such toolbox functions as `NewTrackMedia` and `GetTrackMedia`.
`inputMap`	Specifies the media input map for this operation. You must dispose of the map referred to by this parameter when you are done with it using `QTDisposeAtomContainer`.

DISCUSSION

Use the `GetMediaInputMap` function to specify the media you want to get so you can modify its input map. The caller is responsible for disposing of the input map with `QTDisposeAtomContainer`.

RESULT CODES

`invalidMedia`	–2008	The media is corrupted or invalid
`paramErr`	–50	Invalid parameter specified
`memFullErr`	–108	Not enough room in heap zone

SetMediaInputMap

The `SetMediaInputMap` function replaces the media's existing input map with the given input map.

```
pascal OSErr SetMediaInputMap (
                    Media theMedia,
                    QTAtomContainer inputMap);
```

`theMedia`	Specifies the media for this operation. Your application obtains this media identifier from such toolbox functions as `NewTrackMedia` and `GetTrackMedia`.
`inputMap`	Specifies the media input map for this operation. If the input map is set to `nil`, the media's input map is reset to an empty input map.

DISCUSSION

Use the `SetMediaInputMap` function to specify the media you want to set so you can modify or empty its input map.

`SetMediaInputMap` makes a copy of the input map passed to it. Typically, an application calls `GetMediaInputMap` to get the current input map before modifying it. Use `QTNewAtomContainer` (page 142) to create an empty input map.

RESULT CODES

`invalidMedia`	–2008	The media is corrupted or invalid
`paramErr`	–50	Invalid parameter specified
`memFullErr`	–108	Not enough room in heap zone

Media Functions

Selecting Data Handlers

GetDataHandler

The `GetDataHandler` function allows you to retrieve the best data handler component to use with a given data reference.

```
pascal Component GetDataHandler (Handle dataRef,
                     OSType dataHandlerSubType, long flags);
```

`dataRef` Contains a handle to the data reference. The type of information stored in the handle depends upon the data reference type specified by the `dataHandlerSubType` parameter.

`dataHandlerSubType`
Identifies both the type of data reference and, by implication, the component subtype value assigned to the data handler components that operate on data references of that type.

`flags`	Indicates the way in which you intend to use the data handler component. Note that not all data handlers necessarily support all services—for example, some data handler components may not support streaming writes.

The following flags are defined (set the appropriate flags to 1):

`kDataHCanRead`	Specifies that you intend to use the data handler component to read data.
`kDataHCanWrite`	Specifies that you intend to use the data handler component to write data.
`kDataHCanStreamingWrite`	Indicates that you intend to do streaming writes (as part of a movie-capture operation, for example).

DESCRIPTION

Once you have used this function to get information about the best data handler component for your data reference, you can open and use the component using Component Manager functions. See “Data Handler Components,” for more information.

If the function returns a value of `nil`, the toolbox was unable to find an appropriate data handler component. For more information about the error, call the `GetMoviesError` toolbox function.

RESULT CODES

Memory Manager errors

Timecode Media Handler Functions

This section discusses the functions and structures that allow you to use the timecode media handler.

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

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

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

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

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

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

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

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

You store the timecode's source identification information in a user data item. Create a user data item with a type value of `TCSourceRefNameType`. Store the source information as a text string. This information might contain the name of the videotape from which the movie was created, for example. For more information about working with user data, see *Inside Macintosh: QuickTime.*

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

```
void setTimeCodeSourceName (Media timeCodeMedia,
                                TimeCodeDescriptionHandle tcdH,
                                Str255 tapeName, ScriptCode
tapeNameScript)

{
    UserData srcRef;

    if (NewUserData(&srcRef) == noErr) {
        Handle nameHandle;

        if (PtrToHand(&tapeName[1], &nameHandle, tapeName[0]) == noErr) {
            if (AddUserDataText (srcRef, nameHandle,'name', 1,
                                        tapeNameScript) == noErr) {
                TCSetSourceRef (GetMediaHandler (timeCodeMedia),
                                        tcdH,
                                        srcRef);
            }
            DisposeHandle(nameHandle);
        }
        DisposeUserData(srcRef);
    }
}
```

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

You can define the relationship between a timecode track and one or more movie tracks using the toolbox's new track reference functions (see "Track References" and "Functions for Working With Track References" elsewhere in this chapter for more information). You then proceed to add samples to the track, as appropriate.

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

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

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

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

Field descriptions

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

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

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

Field descriptions

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

`tcDropFrame` Indicates that the timecode "drops" frames occasionally in order to stay in synchronization. Some timecodes run at other than a whole number of frames per second. For example, NTSC video runs at 29.97 frames per second. In order to resynchronize between the timecode

	rate and a 30 frames-per-second playback rate, the timecode drops a frame at a predictable time (in much the same way that leap years keep the calendar synchronized). Set this flag to 1 if the timecode uses the dropframe technique.
`tc24HourMax`	Indicates that the timecode values wrap at 24 hours. Set this flag to 1 if the timecode hour value wraps (that is, returns to 0) at 24 hours.
`tcNegTimesOK`	Indicates that the timecode supports negative time values. Set this flag to 1 if the timecode allows negative values.
`tcCounter`	Indicates that the timecode should be interpreted as a simple counter, rather than as a time value. This allows the timecode to contain either time information or counter (such as a tape counter) information. Set this flag to 1 if the timecode contains counter information.

`fTimeScale`	Contains the time scale for interpreting the `frameDuration` field. This field indicates the number of time units per second.
`frameDuration`	Specifies how long each frame lasts, in the units defined by the `fTimeScale` field.
`numFrames`	Indicates the number of frames stored per second. In the case of timecodes that are interpreted as counters, this field indicates the number of frames stored per timer “tick.”

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

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

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

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

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

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

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

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

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

TCGetCurrentTimeCode

The `TCGetCurrentTimeCode` function retrieves the timecode and source identification information for the current movie time.

```
pascal HandlerError TCGetCurrentTimeCode (MediaHandler mh, long
                     *frameNum, TimeCodeDef *tcdef, TimeCodeRecord
                     *tcrec, UserData *srcRefH);
```

`mh`	Specifies the timecode media handler. You obtain this identifier by calling the `GetMediaHandler` function.
`frameNum`	Contains a pointer to a field that is to receive the current frame number. Set this field to `nil` if you do not want to retrieve the frame number.
`tcdef`	Contains a pointer to a timecode definition structure. The media handler returns the movie's timecode definition information. Set this parameter to `nil` if you do not want this information.
`tcrec`	Contains a pointer to a timecode record structure. The media handler returns the current time value. Set this parameter to `nil` if you do not want this information.
`srcRefH`	Contains a pointer to a field that is to receive a handle containing the source information. It is your responsibility to dispose of this user data when you are done with it. Set this field to `nil` if you do not want this information.

RESULT CODES

`invalidTime`	–2015	This time value is invalid

TCGetTimeCodeAtTime

The `TCGetTimeCodeAtTime` function returns a track's timecode information corresponding to a specific media time.

```
pascal HandlerError TCGetTimeCodeAtTime (MediaHandler mh, TimeValue
                     mediaTime, long *frameNum, TimeCodeDef *tcdef,
                     TimeCodeRecord *tcdata, UserData *srcRefH);
```

`mh`	Specifies the timecode media handler. You obtain this identifier by calling the `GetMediaHandler` function.

`mediaTime`	Specifies the time value for which you want to retrieve timecode information. This time value is expressed in the media's time coordinate system.
`frameNum`	Contains a pointer to a field that is to receive the current frame number. Set this field to `nil` if you do not want to retrieve the frame number.
`tcdef`	Contains a pointer to a timecode definition structure. The media handler returns the movie's timecode definition information. Set this parameter to `nil` if you do not want this information.
`tcrec`	Contains a pointer to a timecode record structure. The media handler returns the current time value. Set this parameter to `nil` if you do not want this information.
`srcRefH`	Contains a pointer to a field that is to receive a handle containing the source information. It is your responsibility to dispose of this user data when you are done with it. Set this field to `nil` if you do not want this information.

RESULT CODES

`invalidTime`	–2015	This time value is invalid

Memory Manager errors

TCTimeCodeToFrameNumber

The `TCTimeCodeToFrameNumber` function converts a timecode time value into its corresponding frame number.

```
pascal HandlerError TCTimeCodeToFrameNumber (MediaHandler mh,
                     TimeCodeDef *tcdef, TimeCodeRecord *tcrec,
                     long *frameNumber);
```

`mh`	Specifies the timecode media handler. You obtain this identifier by calling the `GetMediaHandler` function.
`tcdef`	Contains a pointer to the timecode definition structure to use for the conversion.

`tcrec`	Contains a pointer to the timecode record structure containing the time value to convert.
`frameNumber`	Contains a pointer to a field that is to receive the frame number that corresponds to the time value in the `tcrec` parameter.

RESULT CODES

`paramErr`	–50	Invalid parameter specified

TCFrameNumberToTimeCode

The `TCFrameNumberToTimeCode` function converts a frame number into its corresponding timecode time value.

```
pascal HandlerError TCFrameNumberToTimeCode (MediaHandler mh, long
                     frameNumber, TimeCodeDef *tcdef, TimeCodeRecord
                     *tcrec);
```

`mh`	Specifies the timecode media handler. You obtain this identifier by calling the `GetMediaHandler` function.
`frameNumber`	Specifies the frame number that is to be converted.
`tcdef`	Contains a pointer to the timecode definition structure to use for the conversion.
`tcrec`	Contains a pointer to the timecode record structure that is to receive the time value.

RESULT CODES

`paramErr`	–50	Invalid parameter specified

TCTimeCodeToString

The `TCTimeCodeToString` function converts a time value into a text string (HH:MM:SS:FF). If the timecode uses the dropframe technique, the separators are semicolons (;) rather than colons (:).

```
pascal HandlerError TCTimeCodeToString(MediaHandler mh, TimeCodeDef
                        *tcdef, TimeCodeRecord *tcrec, StringPtr tcStr);
```

`mh`	Specifies the timecode media handler. You obtain this identifier by calling the `GetMediaHandler` function.
`tcdef`	Contains a pointer to the timecode definition structure to use for the conversion.
`tcrec`	Contains a pointer to the timecode record structure to use for the conversion.
`tcStr`	A pointer to a text string that is to receive the converted time value.

RESULT CODES

`paramErr`	–50	Invalid parameter specified

TCSetSourceRef

The `TCSetSourceRef` function allows you to change the source information in the timecode media sample reference.

```
pascal HandlerError TCSetSourceRef (MediaHandler mh,
                        TimeCodeDescriptionHandle tcdH, UserData srefH);
```

`mh`	Specifies the timecode media handler. You obtain this identifier by calling the `GetMediaHandler` function.

`tcdH`	Specifies a handle containing the timecode media sample reference that is to be updated.
`srefH`	Specifies a handle to the source information to be placed in the sample reference. It is your application's responsibility to dispose of this user data when you are done with it.

RESULT CODES

`paramErr`	–50	Invalid parameter specified

Memory Manager errors

TCGetSourceRef

The `TCGetSourceRef` function allows you to retrieve the source information from the timecode media sample reference.

```
pascal HandlerError TCGetSourceRef (MediaHandler mh,
                      TimeCodeDescriptionHandle tcdH, UserData *srefH);
```

`mh`	Specifies the timecode media handler. You obtain this identifier by calling the `GetMediaHandler` function.
`tcdH`	Specifies a handle containing the timecode media sample reference for this operation.
`srefH`	Specifies a pointer to a handle that will receive the source information. It is your application's responsibility to dispose of this user data when you are done with it.

RESULT CODES

`paramErr`	–50	Invalid parameter specified

Memory Manager errors

TCSetTimeCodeFlags

The `TCSetTimeCodeFlags` function allows you to change the flags that affect how the toolbox handles the timecode information.

```
pascal HandlerError TCSetTimeCodeFlags (MediaHandler mh, long flags, long
                         flagsMask);
```

`mh` Specifies the timecode media handler. You obtain this identifier by calling the `GetMediaHandler` function.

`flags` Specifies the new flag values. The following flags are defined:

`tcdfShowTimeCode`
Controls the display of timecode information. Set this flag to 1 to cause timecode information to be displayed when the movie plays. Set this flag to 0 to turn off the display.

Note that the timecode track must be enabled in order for the timecode information to be displayed.

`flagsMask` Specifies which of the flag values are to change. The media handler modifies only those flag values that correspond to bits that are set to 1 in this parameter. Use the flag values from the `flags` parameter. For example, in order to turn off timecode display, you would set the `tcdfShowTimeCode` flag to 1 in the `flagsMask` parameter, and to 0 in the `flags` parameter.

TCGetTimeCodeFlags

The `TCGetTimeCodeFlags` function allows you to retrieve the timecode control flags.

```
pascal HandlerError TCGetTimeCodeFlags (MediaHandler mh, long *flags );
```

`mh` Specifies the timecode media handler. You obtain this identifier by calling the `GetMediaHandler` function.

`flags` Contains a pointer to a field that is to receive the control flags. The following flags are defined:

`tcdfShowTimeCode`
Controls the display of timecode information. If this flag is set to 1, the timecode information is displayed when the movie is played.

Note that the timecode track must be enabled in order for the timecode information to be displayed.

TCSetDisplayOptions

The `TCSetDisplayOptions` function allows you to set the text characteristics that apply to timecode information that is displayed in a movie.

```
pascal HandlerError TCSetDisplayOptions (MediaHandler mh,
                        TCTextOptionsPtr textOptions);
```

`mh`	Specifies the timecode media handler. You obtain this identifier by calling the `GetMediaHandler` function.
`textOptions`	Contains a pointer to a text options structure. This structure contains font and style information.

DESCRIPTION

You provide the text style information in a text options structure. This structure is defined as follows:

```
typedef struct TCTextOptions {
    short       txFont;            /* font */
    short       txFace;            /* font style */
    short       txSize;            /* font size */
    RGBColor    foreColor;         /* foreground color */
    RGBColor    backColor;         /* background color */
} TCTextOptions, *TCTextOptionsPtr;
```

For more information about working with text characteristics, see *Inside Macintosh: Text*.

Field descriptions

`txFont`	Specifies the number of the font.
`txFace`	Specifies the font's style (bold, italic, and so on).
`txSize`	Specifies the font's size.
`foreColor`	Specifies the foreground color.
`backColor`	Specifies the background color.

TCGetDisplayOptions

The `TCGetDisplayOptions` function allows you to retrieve the text characteristics that apply to timecode information that is displayed in a movie.

```
pascal HandlerError TCGetDisplayOptions (MediaHandler mh,
                        TCTextOptionsPtr textOptions);
```

`mh`	Specifies the timecode media handler. You obtain this identifier by calling the `GetMediaHandler` function.
`textOptions`	Contains a pointer to a text options structure. This structure will receive font and style information.

RESULT CODES

`paramErr`	–50	Invalid parameter specified

Media Property Functions

This section discusses functions for setting and retrieving the property atom container of a media handler. For more information about using the sprite media handler with these functions, see "Programming with QuickTime Sprites."

GetMediaPropertyAtom

The `GetMediaPropertyAtom` function retrieves the property atom container of a media handler.

```
pascal OSErr GetMediaPropertyAtom (Media theMedia,
                        QTAtomContainer *propertyAtom)
```

`theMedia`	Contains a reference to the media handler for this operation.
`propertyAtom`	Contains a pointer to a QT atom container. On return, the atom container contains the property atoms for the track associated with the media handler.

DISCUSSION

You can call the `GetMediaPropertyAtom` to retrieve the properties of the track associated with the specified media handler. The contents of the returned QT atom container are defined by the media handler. The caller is responsible for disposing of the QT atom container.

RESULT CODES

`memFullErr`	–108	Not enough room in heap zone
`invalidMedia`	–2008	The media is corrupted or invalid

SetMediaPropertyAtom

The `SetMediaPropertyAtom` function sets the property atom container of a media handler.

```
pascal OSErr SetMediaPropertyAtom (Media theMedia,
                        QTAtomContainer propertyAtom)
```

`theMedia`	Contains a reference to the media handler for this operation.
`propertyAtom`	Specifies a QT atom container that contains the property atoms for the track associated with the media handler.

DISCUSSION

You can call the `SetMediaPropertyAtom` to set properties for the track associated with the specified media handler. The contents of the QT atom container are defined by the media handler.

RESULT CODES

`memFullErr`	–108	Not enough room in heap zone
`invalidMedia`	–2008	The media is corrupted or invalid

Text Media Handler Functions

QuickTime 1.6.1 added five new flags, two constants, and one new function to the text media handler interface. The new flags and constants are defined in "Text Sample Display Flags" (page 74), and "Text Atom Types" (page 78), respectively. The new function is defined in this section.

TextMediaSetTextSampleData

The `TextMediaSetTextSampleData` function allows you to set values before calling the `AddTextSample` or `AddTESample` function.

```
pascal ComponentResult TextMediaSetTextSampleData(
                     MediaHandler mh,
                     void *data,
                     OSType dataType)
```

`mh`	Contains a reference to the text media handler. You obtain this reference from the `GetMediaHandler` function.
`data`	Contains a pointer to the data, defined by the `dataType` parameter.
`dataType`	Specifies the type of data.

RESULT CODES

`memFullErr`	–108	Not enough room in heap zone
`paramErr`	–50	Invalid parameter specified

DISCUSSION

The following sample code demonstrates how to use the `TextMediaSetTextSampleData` function:

```
short           trans = 127;
Point           dropOffset;
MediaHandler    mh;

dropOffset.h = dropOffset.v = 4
TextMediaSetTextSampleData(mh,(void *)&dropOffset,dropShadowOffsetType);
TextMediaSetTextSampleData(mh,(void *)&trans,dropShadowTranslucencyType);
```

Note
Be sure to turn on the `dfDropShadow` display flag after you call `AddTextSample` or `AddTESample`. Passing `nil` for the `textColor` parameter in `AddTextSample` or `AddTESample` defaults to black. Passing `nil` for the `backColor` parameter in `AddTextSample` or `AddTESample` defaults to white. ◆

RESULT CODES

`badComponentInstance`	0x80008001	Invalid component instance specified.

QT Atom Functions

This section describes the functions used to create and manipulate QT atom containers.

Creating and Modifying QT Atom Containers

QTNewAtomContainer

The `QTNewAtomContainer` function allows you to create a new atom container.

```
pascal OSErr QTNewAtomContainer (QTAtomContainer *atomData);
```

`atomData`	Contains a pointer to an unallocated atom container data structure. On return, this parameter points to an allocated atom container.

DISCUSSION

This function creates a new, empty atom container structure. Once you have created an atom container, you can manipulate it using the atom container functions.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memFullError`	–108	Not enough memory available

QTInsertChild

The `QTInsertChild` function creates a new child atom for the specified parent atom.

```
pascal OSErr QTInsertChild (QTAtomContainer container,
                    QTAtom parentAtom,
                    QTAtomType atomType,
                    QTAtomID id,
                    short index,
                    long dataSize,
                    void *data,
                    QTAtom *newAtom);
```

`container`	Specifies the atom container that contains the parent atom. The atom container must not be locked.
`parentAtom`	Specifies the parent atom within the atom container.
`atomType`	Specifies the type of the new atom to be inserted.
`id`	Specifies the ID of the new atom to be inserted. This ID must be unique among atoms of the same type for the specified parent. If you set this parameter to 0, this function assigns a unique ID to the atom.
`index`	Specifies the index of the new atom among atoms with the same parent. To insert the first atom for the specified parent, you should set the `index` parameter to 1. To insert an atom as the last atom in the child list, you should set the `index` parameter to 0.
`dataSize`	Specifies the size of the data for the new atom. If the new atom is to be a parent atom or if you want to add the atom's data later, you should pass 0 for this parameter.
`data`	Contains a pointer to a buffer containing the data for the new atom. If you set the value of the `dataSize` parameter to 0, you should pass `nil` for this parameter.
`newAtom`	Contains a pointer to data of type `QTAtom`. On return, this parameter points to the newly created atom. You can pass `nil` for this parameter if you do not need a reference to the newly created atom.

DISCUSSION

You call this function to create a new child atom. The new child atom has the specified atom type and atom ID, and is inserted into its parent atom's child list at the specified index; any existing atoms at the same index or greater are moved toward the end of the child list. Index values greater than the index of the last atom in the child list plus 1 are invalid.

To create the new atom as a leaf atom that contains data, you should specify the data and its size using the `data` and `dataSize` parameters.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memLockedErr`	–117	Trying to move a locked block
`atomIndexInvalidErr`	–2104	Specified index is out of range
`duplicateAtomTypeAndIDErr`	–2105	An atom with the same type and ID already exists for the specified parent

Memory Manager errors, as documented in *Inside Macintosh: Memory.*

QTInsertChildren

The `QTInsertChildren` function inserts a container of atoms as children of the specified parent atom.

```
pascal OSErr QTInsertChildren (QTAtomContainer container,
                        QTAtom parentAtom,
                        QTAtomContainer childrenContainer);
```

`container` Specifies the atom container that contains the parent atom. The atom container must not be locked.

`parentAtom` Specifies the parent atom within the atom container.

`childrenContainer`
Specifies the atom container that contains the child atoms to be inserted.

DISCUSSION

You call this function to insert a container of atoms as children of a parent atom in another atom container. Each child atom is inserted as the last atom of its type and is assigned a corresponding index. The ID of a child atom to be inserted must not duplicate that of an existing child atom of the same type.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memLockedErr`	–117	Trying to move a locked block
`duplicateAtomTypeAndIDErr`	–2105	An atom with the same type and ID already exists for the specified parent

Memory Manager errors, as documented in *Inside Macintosh: Memory.*

QTReplaceAtom

The `QTReplaceAtom` function replaces the contents of an atom and its children with a different atom and its children.

```
pascal OSErr QTReplaceAtom (QTAtomContainer targetContainer,
                     QTAtom targetAtom,
                     QTAtomContainer replacementContainer,
                     QTAtom replacementAtom);
```

`targetContainer`
Specifies the atom container that contains the atom to be replaced. The atom container must not be locked.

`targetAtom` Specifies the atom to be replaced.

`replacementContainer`
Specifies the atom container that contains the replacement atom.

`replacementAtom`
Specifies the replacement atom.

DISCUSSION

The target atom and the replacement atom must be of the same type. The target atom maintains its original atom ID. This function does not modify the replacement container.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memLockedErr`	–117	Trying to move a locked block
`atomsNotOfSameTypeErr`	–2103	The specified atoms are not of the same type

Memory Manager errors, as documented in *Inside Macintosh: Memory.*

QTSwapAtoms

The `QTSwapAtoms` function swaps the contents of two atoms in an atom container.

```
pascal OSErr QTSwapAtoms (QTAtomContainer container,
                          QTAtom atom1,
                          QTAtom atom2);
```

`container`	Specifies the atom container for this operation.
`atom1`	Specifies an atom to be swapped with the atom specified by `atom2`.
`atom2`	Specifies an atom to be swapped with the atom specified by `atom1`.

DISCUSSION

You call this function to swap the contents of two atoms in an atom container. After swapping, the ID and index of each atom remains the same. The two atoms specified must be of the same type. Either atom may be a leaf atom or a container atom.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`atomNotOfSameTypeErr`	–2103	The specified atoms are not of the same type

Memory Manager errors, as documented in *Inside Macintosh: Memory.*

QTSetAtomID

The `QTSetAtomID` function changes the ID of an atom.

```
pascal OSErr QTSetAtomID (QTAtomContainer container,
                          QTAtom atom,
                          QTAtomID newID);
```

`container`	Specifies the atom container for this operation.
`atom`	Specifies the atom to be modified.
`newID`	Specifies the new ID for the atom.

DISCUSSION

You cannot change an atom's ID to an ID already assigned to a sibling atom of the same type. Also, you cannot change the ID of the container itself by passing 0 for the `atom` parameter.

RESULT CODES

`noErr`	0	No error
`duplicateAtomTypeAndIDErr`	–2105	An atom with the same type and ID already exists for the specified parent
`invalidAtomErr`	–2106	Atom specified by container and offset does not exist, container may be invalid

QTSetAtomData

The `QTSetAtomData` function changes the data of a leaf atom.

```
pascal OSErr QTSetAtomData (QTAtomContainer container,
                          QTAtom atom,
                          long dataSize,
                          void *atomData);
```

`container`	Specifies the atom container that contains the atom to be modified.
`atom`	Specifies the atom to be modified.
`dataSize`	Specifies the length, in bytes, of the data pointed to by the `atomData` parameter.
`atomData`	Contains a pointer to the new data for the atom.

DISCUSSION

You call this function to replace a leaf atom's data with new data. Only leaf atoms contain data; this function returns an error if you pass it to a non-leaf atom.

The atom container specified by the `container` parameter should not be locked. This function may move memory; if the pointer specified by the `atomData` parameter is a dereferenced handle, you should lock the handle.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memLockedErr`	–117	Trying to move a locked block
`notLeafAtomErr`	–2102	Atom specified by container and offset is not a leaf atom

Memory Manager errors, as documented in *Inside Macintosh: Memory.*

QTCopyAtom

The `QTCopyAtom` function copies an atom and its children to a new atom container.

```
pascal OSErr QTCopyAtom (QTAtomContainer container,
                         QTAtom atom,
                         QTAtomContainer *targetContainer);
```

`container`	Specifies the atom container that contains the atom to be copied.
`atom`	Specifies the atom to be copied.

`targetContainer`
Contains a pointer to an uninitialized atom container data structure. On return, this parameter points to an atom container that contains a copy of the atom.

DISCUSSION

To duplicate the entire container specified by the container parameter, you should pass a value of `kParentAtomIsContainer` for the `atom` parameter. The caller is responsible for disposing of the new atom container by calling the `QTDisposeAtomContainer` function.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified

Memory Manager errors, as documented in *Inside Macintosh: Memory.*

QTLockContainer

The `QTLockContainer` function locks an atom container in memory.

```
pascal OSErr QTLockContainer (QTAtomContainer container);
```

`container` Specifies the atom container to be locked.

DISCUSSION

You should call this function to lock an atom container before calling `QTGetAtomDataPtr` to directly access a leaf atom’s data. When you have finished accessing a leaf atom’s data, you should call the `QTUnlockContainer` function.

You may make nested pairs of calls to `QTLockContainer` and `QTUnlockContainer`; you do not need to check the current state of the container first.

RESULT CODES

`noErr`	0	No error
`invalidAtomContainerErr`	–2107	Specified atom container is invalid

QTGetAtomDataPtr

The `QTGetAtomDataPtr` function retrieves a pointer to the atom data for the specified leaf atom.

```
pascal OSErr QTGetAtomDataPtr (QTAtomContainer container,
                          QTAtom atom,
                          long *dataSize,
                          Ptr *atomData);
```

`container`	Specifies the atom container that contains the leaf atom.
`atom`	Specifies the leaf atom whose data should be retrieved.
`dataSize`	On return, contains a pointer to the length, in bytes, of the leaf atom's data.
`atomData`	On return, contains a pointer to the leaf atom's data.

DISCUSSION

You call this function in retrieve a pointer to a leaf atom's data so that you can access the data directly. To ensure that the pointer returned in the `atomData` parameter will remain valid if memory is moved, you should call `QTLockContainer` before you call this function. If you call `QTLockContainer`, you should call `QTUnlockContainer` when you have finished using the `atomData` pointer; if you pass a locked atom container to a function that resizes atom containers, the function returns an error.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`notLeafAtomErr`	–2102	Atom specified by container and offset is not a leaf atom

QTUnlockContainer

The `QTLockContainer` function unlocks an atom container in memory.

```
pascal OSErr QTUnlockContainer (QTAtomContainer container);
```

`container`	Specifies the atom container to be unlocked.

DISCUSSION

You should call this function to unlock an atom container when you have finished accessing a leaf atom's data.

You may make nested pairs of calls to `QTLockContainer` and `QTUnlockContainer`; you do not need to check the current state of the container first.

RESULT CODES

`noErr`	0	No error
`invalidAtomContainerErr`	–2107	Specified atom container is invalid

QTRemoveAtom

The `QTRemoveAtom` function removes an atom and its children from the specified atom container.

```
pascal OSErr QTRemoveAtom (QTAtomContainer container,
                           QTAtom atom);
```

`container`	Specifies the atom container for this operation. The atom container must not be locked.
`atom`	Specifies the atom to be removed from the container.

DISCUSSION

You call this function to remove a particular atom and its children from an atom container. To remove all the atoms in an atom container, you should use the `QTRemoveChildren` function.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memLockedErr`	–117	Trying to move a locked block

Memory Manager errors, as documented in *Inside Macintosh: Memory.*

QTRemoveChildren

The `QTRemoveChildren` function removes all the children of an atom from the specified atom container.

```
pascal OSErr QTRemoveChildren (QTAtomContainer container,
                               QTAtom atom);
```

`container`	Specifies the atom container for this operation. The atom container must not be locked.
`atom`	Specifies the atom whose children should be removed.

DISCUSSION

To remove all the atoms in the atom container, pass a value of `kParentAtomIsContainer` for the `atom` parameter.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memLockedErr`	–117	Trying to move a locked block

Memory Manager errors, as documented in *Inside Macintosh: Memory.*

QTDisposeAtomContainer

The `QTDisposeAtomContainer` function disposes of an atom container.

```
pascal OSErr QTDisposeAtomContainer (QTAtomContainer atomData);
```

`atomData`	Specifies the atom container to be disposed of.

DISCUSSION

You can call this function to dispose of an atom container data structure that was created by `QTNewAtomContainer` or `QTCopyAtom`.

RESULT CODES

`noErr`	0	No error
`invalidAtomContainerErr`	–2107	Specified atom container is invalid

Retrieving Atoms and Atom Data

QTGetNextChildType

The `QTGetNextChildType` function returns the next atom type in the child list of the specified parent atom.

```
pascal QTAtomType QTGetNextChildType (QTAtomContainer container,
                    QTAtom parentAtom,
                    QTAtomType currentChildType);
```

`container`	Specifies the atom container that contains the parent atom.
`parentAtom`	Specifies the parent atom for this operation.
`currentChildType`	Specifies the last atom type retrieved by this function.
function result	The atom type.

DISCUSSION

You can call this function to iterate through the atom types in a parent atom's child list. To retrieve the first atom type, you should set the value of the `currentChildType` parameter to 0. To retrieve subsequent atom types, you should set the value of the `currentChildType` parameter to the atom type retrieved by the previous call to this function.

QTCountChildrenOfType

The `QTCountChildrenOfType` function returns the number of atoms of a given type in the child list of the specified parent atom.

```
pascal short QTCountChildrenOfType (QTAtomContainer container,
                        QTAtom parentAtom,
                        QTAtomType childType);
```

`container`	Specifies the atom container that contains the parent atom.
`parentAtom`	Specifies the parent atom for this operation.
`childType`	Specifies the atom type for this operation.
function result	The number of atoms of the specified type in the parent atom's child list.

DISCUSSION

You can call this function to determine the number of atoms of a specified type in a parent atom's child list. To retrieve the total number of atoms in the child list, you should set the value of the `childType` parameter to 0. If the total number of atoms in the parent atom's child list is 0, the parent atom is a leaf atom.

QTFindChildByIndex

The `QTFindChildByIndex` function retrieves an atom by index from the child list of the specified parent atom.

```
pascal QTAtom QTFindChildByIndex (QTAtomContainer container,
                         QTAtom parentAtom,
                         QTAtomType atomType,
                         short index,
                         QTAtomID *id);
```

`container`	Specifies the atom container that contains the parent atom.
`parentAtom`	Specifies the parent atom for this operation.
`atomType`	Specifies the type of the atom to be retrieved.
`index`	Specifies the index of the atom to be retrieved.
`id`	Contains a pointer to an uninitialized `QTAtomID` data structure. On return, if the atom specified by `index` was found, the `QTAtomID` data structure contains the atom's ID.
function result	The child atom, if found; otherwise, 0.

DISCUSSION

You call this function to search for and retrieve an atom by its type and index within that type from a parent atom's child list. If you do not want this function to return the atom's ID, set the value of the `id` parameter to `nil`.

QTFindChildByID

The `QTFindChildByID` function retrieves an atom by ID from the child list of the specified parent atom.

```
pascal QTAtom QTFindChildByID (QTAtomContainer container,
                         QTAtom parentAtom,
                         QTAtomType atomType,
                         QTAtomID id,
                         short *index);
```

`container`	Specifies the atom container that contains the parent atom.
`parentAtom`	Specifies the parent atom for this operation.
`atomType`	Specifies the type of the atom to be retrieved.

`id`	Specifies the ID of the atom to be retrieved.
`index`	Contains a pointer to an uninitialized short integer. On return, if the atom specified by `id` was found, the integer contains the atom's index.
function result	The child atom, if found; otherwise, 0.

DISCUSSION

You call this function to search for and retrieve an atom by its type and ID from a parent atom's child list. If you do not want this function to return the atom's index, set the value of the `index` parameter to `nil`.

QTNextChildAnyType

The `QTNextChildAnyType` function returns the next atom in the child list of the specified parent atom.

```
pascal OSErr QTNextChildAnyType (QTAtomContainer container,
                        QTAtom parentAtom,
                        QTAtom currentChild,
                        QTAtom *nextChild);
```

`container`	Specifies the atom container that contains the parent atom.
`parentAtom`	Specifies the parent atom for this operation.
`currentChild`	Specifies the last atom retrieved by this function.
`nextChild`	Contains a pointer to an uninitialized `QTAtom` data structure. On return, the data structure contains the offset of the next atom in the child list after the atom specified by `currentChild`, or 0 if the atom specified by `currentChild` was the last atom in the list.

DISCUSSION

You can call this function to iterate through all the atoms in a parent atom's child list, regardless of their types and IDs. To retrieve the first atom in the child list, set the value of the `currentChild` parameter to 0.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`invalidAtomErr`	–2106	Atom specified by container and offset does not exist, container may be invalid

QTCopyAtomDataToHandle

The `QTCopyAtomDataToHandle` function copies the specified leaf atom's data to a handle.

```
pascal OSErr QTCopyAtomDataToHandle (QTAtomContainer container,
                      QTAtom atom,
                      Handle targetHandle);
```

`container`	Specifies the atom container that contains the leaf atom.
`atom`	Specifies the leaf atom whose data should be copied.
`targetHandle`	Contains a handle. On return, the handle contains the atom's data. The handle must not be locked.

DISCUSSION

You call this function, passing an initialized handle, to retrieve a copy of a leaf atom's data. This function resizes the handle, if necessary.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memLockedErr`	–117	Trying to move a locked block
`notLeafAtomErr`	–2102	Atom specified by container and offset is not a leaf atom

Memory Manager errors, as documented in *Inside Macintosh: Memory.*

QTCopyAtomDataToPtr

The `QTCopyAtomDataToPtr` function copies the specified leaf atom's data to a buffer.

```
pascal OSErr QTCopyAtomDataToPtr (QTAtomContainer container,
                    QTAtom atom,
                    Boolean sizeOrLessOK,
                    long size,
                    void *targetPtr,
                    long *actualSize);
```

`container`	Specifies the atom container that contains the leaf atom.
`atom`	Specifies the leaf atom whose data should be copied.
`sizeOrLessOK`	Specifies whether the function may copy fewer bytes than the number of bytes specified by the `size` parameter.
`size`	Specifies the length, in bytes, of the buffer pointed to by the `targetPtr` parameter.
`targetPtr`	Contains a pointer to a buffer. On return, the buffer contains the atom data.
`actualSize`	Contains a pointer to a long integer which, on return, contains the number of bytes copied to the buffer.

DISCUSSION

You call this function, passing a data buffer, to retrieve a copy of a leaf atom's data. The buffer must be large enough to contain the atom's data. The buffer may be larger than the amount of atom data if you set the value of the `sizeOrLessOK` parameter to `true`. You can determine the size of an atom's data by calling `QTGetAtomDataPtr`.

This function may move memory.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`notLeafAtomErr`	–2102	Atom specified by container and offset is not a leaf atom

Memory Manager errors, as documented in *Inside Macintosh: Memory.*

QTGetAtomTypeAndID

The `QTGetAtomTypeAndID` function retrieves an atom's type and ID.

```
pascal OSErr QTGetAtomTypeAndID (QTAtomContainer container,
                    QTAtom atom,
                    QTAtomType *atomType,
                    QTAtomID *id);
```

`container`	Specifies the atom container that contains the atom.
`atom`	Specifies the atom whose type and ID should be retrieved.
`atomType`	Contains a pointer to an atom type. On return, this parameter points to the type of the specified atom. You can pass `nil` for this parameter if you do not need this information.
`id`	Contains a pointer to an atom ID. On return, this parameter points to the ID of the specified atom. You can pass `nil` for this parameter if you do not need this information.

DISCUSSION

You call this function to retrieve the type and ID for a particular atom.

RESULT CODES

`noErr`	0	No error
`invalidAtomErr`	–2106	Atom specified by container and offset does not exist, container may be invalid

Access Key Functions

QTGetAccessKeys

The `QTGetAccessKeys` function returns all the application and system access keys of a specified access key type.

```
pascal OSErr QTGetAccessKeys (Str255 accessKeyType, long flags,
      QTAtomContainer *keys);
```

`accessKeyType`	The type of access keys to return.
`flags`	Unused; must be set to 0.
`keys`	A pointer to a QT atom container that contains the keys.

DISCUSSION

The QT atom container returned by this function contains atoms of type `kAccessKeyAtomType` at the top level. These atoms contain the keys. You can get the key values by using QT atom functions described in “QuickTime Atoms” (page 47). In the QT atom container, application keys (which are more likely to be the ones an application needs) appear before system keys.

If there are no access keys of the specified type, the function returns an empty QT atom container.

When your software is done with the QT atom container, it must dispose of it by calling the `QTDisposeAtomContainer` function.

QTRegisterAccessKey

The `QTRegisterAccessKey` function registers an access key.

```
pascal OSErr QTRegisterAccessKey (Str255 accessKeyType, long flags,
      Handle accessKey);
```

`accessKeyType`	The access key type of the key to be registered.
`flags`	Flags that specify the operation of this function. To register a system access key, set the `kAccessKeySystemFlag` flag. To register an application access key, set this parameter to 0.
`accessKey`	The key to be registered.

DISCUSSION

Most access keys are strings. A string stored in the `accessKey` handle does not include a trailing zero or leading length byte; the length of the string is the size of the handle.

If the access key has already been registered, no error is returned, and the request is simply ignored.

QTUnregisterAccessKey

The `QTUnregisterAccessKey` function removes a previously registered access key.

```
pascal OSErr QTUnregisterAccessKey (Str255 accessKeyType, long flags,
      Handle accessKey);
```

`accessKeyType`	The access key type of the key to be removed.
`flags`	Flags that specify the operation of this function. To remove a system access key, set the `kAccessKeySystemFlag` flag. To remove an application access key, set this parameter to 0.
`accessKey`	The key to be removed.

DISCUSSION

Most access keys are strings. A string stored in the `accessKey` handle does not include a trailing zero or leading length byte.

Functions for Progressive Downloads

High-Level Routines

The Movie Toolbox includes support for progressive downloads, which allow part of a movie to be displayed before all of its data has been received over a network or other slow link. Applications that use the movie controller component provided by Apple automatically get support for progressive downloads. Applications that do not use the standard move controller can use the two high-level functions for progressive downloads described in this section, `QTMovieNeedsTimeTable` and `GetMaxLoadedTimeInMovie`, to determine whether a movie is being progressively downloaded and, if so, to see how much of it has already been downloaded.

GetMaxLoadedTimeInMovie

When a movie is being progressively downloaded, the `GetMaxLoadedTimeInMovie` function returns the duration of the part of a movie that has already been downloaded.

```
pascal OSErr GetMaxLoadedTimeInMovie (Movie theMovie,
      TimeValue *time);
```

`theMovie`	Specifies the movie for this operation. Your application obtains this identifier from such toolbox functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle`.
`time`	The duration of the part of a movie that has already been downloaded. This time value is expressed in the movie's time coordinate system.

DISCUSSION

If all of a movie has been downloaded, the `GetMaxLoadedTimeInMovie` function returns the duration of the entire movie.

The toolbox creates a time table for a movie when either `QTMovieNeedsTimeTable` or `GetMaxLoadedTimeInMovie` is called for the movie, but the time table is used only by the toolbox and is not accessible to applications. The toolbox disposes of the time table when the download is complete.

QTMovieNeedsTimeTable

The `QTMovieNeedsTimeTable` function returns whether a movie is being progressively downloaded.

```
pascal OSErr QTMovieNeedsTimeTable (Movie theMovie,
      Boolean *needsTimeTable);
```

`theMovie`	Specifies the movie for this operation. Your application obtains this identifier from such toolbox functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle`.
`needsTimeTable`	If `true`, the movie is being progressively downloaded. If an error occurs, this parameter is set to `false`.

DISCUSSION

A movie can be progressively downloaded when its data is received over a network connection or other slow data channel. Progressive downloads are not necessary when the data for the movie is on a local disk.

The toolbox creates a time table for a movie when either `QTMovieNeedsTimeTable` or `GetMaxLoadedTimeInMovie` is called for the movie, but the time table is used only by the toolbox and is not accessible to applications. The toolbox disposes of the time table when the download is complete.

Low-Level Routines

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

MakeMediaTimeTable

The `MakeMediaTimeTable` function returns a time table for the specified media.

```
pascal OSErr MakeMediaTimeTable (
      Media theMedia,
      long **offsets,
      TimeValue startTime,
      TimeValue endTime,
      TimeValue timeIncrement,
      short firstDataRefIndex,
      short lastDataRefIndex,
      long *retdataRefSkew);
```

`theMedia`	Specifies the media for this operation. Your application obtains this identifier from such toolbox functions as `NewTrackMedia` and `GetTrackMedia`.
`offsets`	A handle to an unlocked relocatable memory block allocated by your application. The function returns the time table for the media in this block.
`startTime`	Specifies the first point of the media to be included in the time table. This time value is expressed in the media's time coordinate system.
`endTime`	Specifies the last point of the media to be included in the time table. This time value is expressed in the media's time coordinate system.
`timeIncrement`	The resolution of the time table. The values in a time table are for a points in the media, and these points are separated by the amount of time specified by this parameter. The time value is expressed in the media's time coordinate system.
`firstDataRefIndex`	An index to the first data reference for the media to be included in the time table. Set this parameter to –1 to include all data references for the media. Set this parameter to 1 to specify the first data reference for the media.

`lastDataRefIndex`
: An index to the last data reference for the media to be included in the time table. The value `1` specifies the first data reference for the media. If the value of the `firstDataRefIndex` parameter is –1, set this parameter to 0.

`retdataRefSkew`
: The offset to the next row of the time table, in long integers. The next row contains values for the next data reference, as explained below. By adding the value of `retdataRefSkew` to an offset into the table, you get the offset to the corresponding point for the next data reference.

DISCUSSION

Your application must allocate an unlocked relocatable memory block for the time table to be returned and pass a handle to it in the `offsets` parameter. The `MakeMediaTimeTable` function resizes the block to accommodate the time table it returns.

The time table returned by the `MakeMediaTimeTable` function is a two-dimensional array of long integers, organized as follows:

- Each row in the table contains values for one data reference.
- The first column in the table contains values for the time in the media specified by the `startTime` parameter, and each subsequent column contains values for the point in the media that is later by the value specified by the `timeIncrement` parameter.
- Each long integer value in the table specifies the offset, in bytes, from the beginning of the data reference for that point in the media.

The number of columns in the table is equal to (`endTime - startTime`) / `timeIncrement`, rounded up. Because of alignment issues, this value is not always the same as the value of the `retdataRefSkew` parameter.

When all the data for a movie has been transferred, your application must dispose of the time table created by this function.

MakeTrackTimeTable

The `MakeTrackTimeTable` function returns a time table for a specified track in a movie.

```
pascal OSErr MakeTrackTimeTable(
      Track trackH,
      long **offsets,
      TimeValue startTime,
      TimeValue endTime,
      TimeValue timeIncrement,
      short firstDataRefIndex,
      short lastDataRefIndex,
      long *retdataRefSkew);
```

`trackH`	Specifies the track for the operation. Your application gets this identifier from such toolbox functions as `NewMovieTrack` and `GetMovieTrack`.
`offsets`	A handle to an unlocked relocatable memory block allocated by your application. The function returns the time table for the track in this block.
`startTime`	Specifies the first point of the track to be included in the time table. This time value is expressed in the movie's time coordinate system.
`endTime`	Specifies the last point of the track to be included in the time table. This time value is expressed in the movie's time coordinate system.
`timeIncrement`	The resolution of the time table. The values in a time table are for a points in the track, and these points are separated by the amount of time specified by this parameter. The time value is expressed in the movie's time coordinate system.
`firstDataRefIndex`	An index to the first data reference for the track to be included in the time table. Set this parameter to –1 to include all data references for the track. Set this parameter to 1 to specify the first data reference for the track.

`lastDataRefIndex`
An index to the last data reference for the track to be included in the time table. The value 1 specifies the first data reference for the track. If the value of the `firstDataRefIndex` parameter is –1, set this parameter to 0.

`retdataRefSkew`
The offset to the next row of the time table, as a long integer. The next row contains values for the next data reference, as explained below. By adding the value of `retdataRefSkew` to an offset into the table, you get the offset to the corresponding point for the next data reference.

DISCUSSION

Your application must allocate an unlocked relocatable memory block for the time table to be returned and pass a handle to it in the `offsets` parameter. The `MakeTrackTimeTable` function resizes the block to accommodate the time table it returns.

The time table returned by the `MakeTrackTimeTable` function is a two-dimensional array of long integers that is organized as follows:

- Each row in the table contains values for one data reference.
- The first column in the table contains values for the time in the track specified by the `startTime` parameter, and each subsequent column contains values for the point in the track that is later by the value specified by the `timeIncrement` parameter.
- Each long integer value in the table specifies the offset, in bytes, from the beginning of the data reference for that point in the track.

The number of columns in the table is equal to (`endTime` - `startTime`) / `timeIncrement`, rounded up. Because of alignment issues, this value is not always the same as the value of the `retdataRefSkew` parameter.

If there are track edits for a track, they are reflected in the track's time table.

When all the data for a movie has been transferred, your application must dispose of the time table created by this function.

C H A P T E R 2

Component Manager

This chapter discusses changes to the Component Manager as documented in Chapter 6 of *Mac OS For QuickTime Programmers*.

New Features of the Component Manager

New Component Manager Functions

The Component Manager includes these new functions: `GetComponentTypeModSeed`, `OpenAComponent`, `OpenADefaultComponent`, `OpenAComponentResFile`, and `CallComponentFunctionWithStorageProcInfo`. The `GetComponentTypeModSeed` function is similar to the `GetComponentListModSeed` function. The `OpenAComponent`, `OpenADefaultComponent` and `OpenAComponentResFile` functions expand upon existing Component Manager routines by adding an error return value. The `CallComponentFunctionWithStorageProcInfo` function is used for PowerPC-native component dispatching.

PowerPC-Native Component Manager Support

The Component Manager was enhanced as part of QuickTime 2.5 to better support PowerPC-native components.

On Mac OS computers, the Component Manager dispatcher for calling components is now fat, avoiding the overhead of the mixed mode switches through the old 68K Component Manager dispatch for native-native component calls. `DelegateComponentCall` is also fat, so native-native delegations avoid the mixed mode switch.

In addition, writing the component dispatch routine for native components has been made significantly easier with the addition of `CallComponentFunctionWithStorageProcInfo`. This call is analogous to `CallComponentFunctionWithStorage`, with the addition of a parameter to pass the procedure information for the desired function. This allows the Mixed Mode Manager to correctly dispatch the call without your code having to unravel the parameters from the `ComponentParameters` block yourself.

To use the new `CallComponentFunctionWithStorageProcInfo` call, your component needs to link with `ComponentsInterfacesLib`.

An example component that uses this technique follows. The component supports the required suite of Open, Close, CanDo, and Version calls, as well as a Beep call (selector 0). The `ExampleComponentDispatch` and `ExampleFindRoutineProcPtr` routines provide the dispatching using the new `CallComponentFunctionWithStorageProcInfo` call.

```
#include <Sound.h>
#include <Components.h>

pascal ComponentResult ExampleComponentDispatch
        (ComponentParameters *params, Handle storage);

static ProcPtr ExampleFindRoutineProcPtr
        (short selector, ProcInfoType *procInfo);

pascal ComponentResult
        ExampleCanDo(Handle storage, short selector);

pascal ComponentResult
        ExampleOpen(Handle storage, ComponentInstance self);

pascal ComponentResult
        ExampleClose(Handle storage,ComponentInstance self);

pascal ComponentResult
        ExampleVersion(Handle storage);

pascal ComponentResult
        ExampleBeep(Handle storage);

#ifdef GENERATINGPOWERPC
```

```
struct RoutineDescriptor ExampleComponentDispatchRD =
        BUILD_ROUTINE_DESCRIPTOR(
            (kPascalStackBased | RESULT_SIZE (kFourByteCode) |
                STACK_ROUTINE_PARAMETER (1, kFourByteCode) |
                STACK_ROUTINE_PARAMETER (2, kFourByteCode)),
            ExampleComponentDispatch);
#endif

enum {
    kExampleBeepSelect = 0
};

enum {
    uppExampleBeepProcInfo = kPascalStackBased
         | RESULT_SIZE(SIZE_CODE(sizeof(ComponentResult)))
         | STACK_ROUTINE_PARAMETER(1, SIZE_CODE(sizeof(Handle)))
};

pascal ComponentResult
ExampleComponentDispatch(ComponentParameters *params, Handle storage)
{
    ProcPtr theProc;
    ProcInfoType theProcInfo;
    ComponentResult result = codecUnimpErr;
    theProc = ExampleFindRoutineProcPtr(params->what, &theProcInfo);
    if (theProc)
        result = CallComponentFunctionWithStorageProcInfo(
                (Handle)storage, params, theProc, theProcInfo);
    return result;
}

static ProcPtr
ExampleFindRoutineProcPtr(short selector, ProcInfoType *procInfo)
{
    ProcPtr aProc;
    ProcInfoType pi;

#define ComponentCall(a)\
        case kComponent##a##Select:  \
            aProc = (ProcPtr)Example##a; \
            pi = uppCallComponent##a##ProcInfo; break;
#define ExampleCall(a)\
        case kExample##a##Select: \
            aProc = (ProcPtr)Example##a; \
            pi = uppExample##a##ProcInfo; break;
    switch (selector) {
        ComponentCall(Version)
        ComponentCall(CanDo)
        ComponentCall(Close)
        ComponentCall(Open)
        ExampleCall(Beep)

        default:
            aProc = nil;
            pi = 0;
        }
    *procInfo = pi;
    return aProc;
}

pascal ComponentResult ExampleCanDo(Handle storage, short selector)
{
    ProcInfoType ignoreResult;
    return (ExampleFindRoutineProcPtr(selector,&ignoreResult) != 0);
}

pascal ComponentResult ExampleOpen(Handle storage,
    ComponentInstance self)
{
    return noErr;
}

pascal ComponentResult ExampleClose(Handle storage,
    ComponentInstance self)
{
    return(noErr);
}

pascal ComponentResult ExampleVersion(Handle storage)
{
    return 0x00010001;
}
```

```
pascal ComponentResult ExampleBeep(Handle storage)
{
    SysBeep(2);
    return noErr;
}
```

Component Aliases

QuickTime 3 supports a new type of component, known as a **component alias,** that is a reference to another component. A component alias does not contain code. The component it refers to, known as the **target component,** contains the necessary code.

If there are several related components that use the same code, such as importers for related file types, you can use component aliases to share the code rather than duplicating it for each component. Each of the component aliases can have its own subtype, manufacturer, name, icon, and other characteristics, even though it shares code with other components.

Contents of a Component Alias

A component alias is contained in a resource of type `kComponentAliasResourceType`. This resource, which is described in Appendix B, contains a component alias resource data structure:

```
struct ComponentAliasResource {
    ComponentResource      cr;  /* registration parameters */
    ComponentDescription   aliasCD;/* target description */
};
```

The `cr` field is a `ComponentResource` structure that contains information needed to register the component. This information is used in the same way as registration information for ordinary components, with one exception: the specification of the code resource for the component is ignored.

The `aliasCD` field is a `ComponentDescription` structure that specifies the target for the alias. The next section, “How a Component Alias Is Resolved” (page 174), explains how this structure is used to find the target. The code resource of the target is used in place of the code resource specified in the `cr` field of the `ComponentAliasResource` structure.

How a Component Alias Is Resolved

To resolve a component alias, the Component Manager passes the contents of the `aliasCD` field of the `ComponentAliasResource` structure to the `FindNextComponent` function, just as an application would. This field includes all the information that is necessary to specify the target component. If there is no component that matches the specifications, the request to open the alias fails.

The Component Manager never resolves a component alias until the alias is opened. This lets you register component aliases before their target components are registered. It also lets you replace target components with other components.

Scope of a Component Alias

Like other components, a component alias has a scope that determines which applications can use its services. The scope of a component alias is determined by the scope of its target component. If its scope is global, its services are available to all applications. If its scope is local, its services are available only to the application that registers it.

Using the Component Manager

The following sections explain how to use new features of the Component Manager that are described in this guide.

Using Component Aliases

The following sections explain how to create and manage component aliases.

Registering a Component Alias

The Component Manager automatically registers component resources stored in the Extensions folder. After it registers all of the component resources in the Extensions folder, it then automatically registers any component alias resources stored in the folder.

To register a component alias whose resource file is stored in a folder other than the Extensions folder, or to specify whether the scope of a component alias is global or local, use the `RegisterComponentResourceFile` function.

Once a component alias is registered, applications can find the component and retrieve information about it using the Component Manager routines described in the "Component Manager" chapter of *Mac OS For QuickTime Programmers* and in this chapter.

Getting Information About a Component Alias

When your application passes a component alias to the `GetComponentInfo` function, it gets information about an alias rather its target.

Opening the Resource File for a Target or Alias Component

When your application passes a component alias to the `OpenComponentResFile` function, the target component's resource file is opened. This ensures that the target component, which is the one that is processing requests, has access to its own resource fork. To open the component alias's resource file, pass the component alias to the `OpenComponentResourceFile` function.

Component Manager Reference

Constants

The Component Alias Resource Type

```
enum {
    kComponentAliasResourceType= FOUR_CHAR_CODE('thga')/* component alias
resource type */
};
```

Constant description

`kComponentAliasResourceType`	The resource type of a component alias resource.

Data Types

The Component Alias Resource Structure

The `ComponentAliasResource` structure is defined as follows:

```
struct ComponentAliasResource {
    ComponentResource      cr;  /* registration parameters */
    ComponentDescription   aliasCD;/* target description */
};
```

Field descriptions

`cr`	Contains information needed to register the component. This information is used in the same way as registration information for ordinary components, with one exception: the specification of the code resource for the component is ignored.
`aliasCD`	Specifies the target for the alias. To resolve a component alias, the Component Manager passes the contents of this field to the `FindNextComponent` function. The code resource of the target is used in place of the code resource specified in the `cr` field.

Functions

Dispatching to Component Routines

CallComponentFunctionWithStorageProcInfo

To use the new `CallComponentFunctionWithStorageProcInfo` function, your component will need to link with `ComponentsInterfacesLib`.

```
pascal long CallComponentFunctionWithStorageProcInfo(
                    Handle storage,
                    ComponentParameters *params,
                    ProcPtr func,
                    long funcProcInfo);
```

`storage`	A handle to the memory associated with the current connection. The Component Manager provides this handle to your component along with the request.
`params`	The component parameters record that your component received from the Component Manager.
`func`	The address of the function that is to handle the request. The Component Manager calls the routine referred to by the `func` parameter as a Pascal function with the parameters that were originally provided by the application. These parameters are preceded by a handle to the memory associated with the current connection. The routine referred to by the `func` parameter must return a function result of type `ComponentResult` (a long integer) indicating the success or the failure of the operation. Note that for PowerPC code, the `func` parameter should still point to the routine itself—not to a `RoutineDescriptor` or Universal Procedure Pointer.

`funcProcInfo` The procedure information for the routine referred to by the `func` parameter. See *Inside Macintosh: PowerPC System Software,* pages 2-14 through 2-21 for information on procedure information data.

Finding Components

GetComponentTypeModSeed

The `GetComponentTypeModSeed` function allows you to determine if the specified type of registered component has changed. This function returns the value of the component registration seed number for the specified component type. By comparing this value to values previously returned by this function, you can determine whether the component registry for the specified type has changed.

```
pascal long GetComponentTypeModSeed (OSType componentType);
```

`componentType`
A four-character code that identifies the type of component. All components of a particular type support a common set of interface routines. Your application uses this field to search for components of a given type.

function result
Returns a long integer containing the component registration seed number. Each time the Component Manager registers or unregisters a component, it generates a new, unique seed number.

DISCUSSION

This function is similar to the `GetComponentListModSeed` function. Unlike `GetComponentListModSeed`, the `GetComponentTypeModSeed` function is specific to a single component type. This allows you to know if, for example, the registration of image decompressor (‘`imdc`’) components has changed regardless of other component changes.

Opening and Closing Components

OpenAComponent

The `OpenAComponent` function is similar to `OpenComponent`, except that its return value is an `OSErr`. The `ComponentInstance` of the newly opened component is passed back through the `ci` argument.

```
pascal OSErr OpenAComponent (
                     Component aComponent,
                     ComponentInstance *ci);
```

`aComponent`	A component identifier that specifies the component to open. Your application obtains this identifier from the `FindNextComponent` function. If your application registers a component, it can also obtain a component identifier from the `RegisterComponent` or `RegisterComponentResource` function.
`ci`	A pointer to a field to receive the `ComponentInstance` of the newly-opened component.

OpenADefaultComponent

The `OpenADefaultComponent` function is similar to `OpenDefaultComponent`, except that its return value is an `OSErr`. The `ComponentInstance` of the newly opened component is passed back through the `ci` argument.

```
pascal OSErr OpenADefaultComponent (
                     OSType componentType,
                     OSType componentSubType,
                     ComponentInstance *ci);
```

`componentType`
: A four-character code that identifies the type of component. All components of a particular type support a common set of interface routines. Your application uses this field to search for components of a given type.

`componentSubType`
: A four-character code that identifies the subtype of the component. Different subtypes of a component type may support additional features or provide interfaces that extend beyond the standard routines for a given component type. For example, the subtype of an image compressor component indicates the compression algorithm employed by the compressor.

  Your application can use the `componentSubType` field to perform a more specific lookup operation than is possible using only the `componentType` field. For example, you may want your application to use only components of a certain component type (‘`draw`’) that also have a specific subtype (‘`oval`’). Set this parameter to 0 to select a component with any subtype value.

`ci`
: A pointer to a field to receive the `ComponentInstance` of the newly-opened component.

Accessing a Component’s Resource File

OpenAComponentResFile

The `OpenAComponentResFile` function is similar to `OpenComponentResFile`, except that its return value is an `OSErr`. The resource reference number of the newly opened component file is passed back through the `resRef` argument.

```
pascal OSErr OpenAComponentResFile (
                    Component aComponent,
                    short *resRef);
```

`aComponent`	A component identifier that specifies the component whose resource file you want to open. Your application can obtain this identifier from the `RegisterComponentResource` or `FindNextComponent` functions, or it can be a `ComponentInstance` (in which case it's the result of `OpenAComponent` or `OpenADefaultComponent`).
`resRef`	A pointer to a field to receive the resource reference number of the newly opened component resource file.

Image Compression Manager

This chapter discusses new features and changes to the Image Compression Manager as documented in Chapter 3 of *Inside Macintosh: QuickTime*.

New Features of the Image Compression Manager

ColorSync Support

ColorSync is a system extension that provides a platform for consistent color reproduction between widely varying output devices. ColorSync color matching capability was added to the Image Compression Manager picture drawing functions in QuickTime 1.6.1. You can now accurately reproduce color images (not movies) with the `DrawPicture` functions by setting the `useColorMatching` flag in the flags parameter to these functions.

```
enum {
    useColorMatching    = 4
};
```

For more information about QuickTime picture drawing functions, see "Working With Pictures and PICT Files," beginning on page 3-88 of *Inside Macintosh: QuickTime.*

Asynchronous Decompression

QuickTime 2.0 introduced the concept of scheduled asynchronous decompression operations. Decompressor components can allow applications to queue decompression operations and specify when those operations should take place. The Image Compression Manager provides a new function, `DecompressSequenceFrameWhen` (page 193), that allows applications to schedule an asynchronous decompression operation.

Timecode Support

Timecode tracks were introduced in QuickTime 2.0. The Image Compression Manager and compressor components have been enhanced to support timecode information. The Image Compression Manager function `SetDSequenceTimeCode` (page 200) allows you to set the timecode value for a frame that is to be decompressed. For more information about timecode tracks and the timecode media handler, see Chapter 1, “Movie Toolbox.”

Data Source Support

QuickTime 2.1 introduced support for an arbitrary number of sources of data for an image sequence. This functionality forms the basis for dynamically modifying parameters to a decompressor. It also allows for codecs to act as special effects components, providing filtering and transition type effects. A client can attach an arbitrary number of additional inputs to the codec. It is up to the particular codec to determine whether to use each input and how to interpret the input. For example, an 8-bit gray image could be interpreted as a blend mask or as a replacement for one of the RGB data planes.

To create a new data source, use the function `CDSequenceNewDataSource` (page 206).

Working with Alpha Channels

QuickTime has always supported compressing and storing images with an alpha channel. In QuickTime 2.5, the Image Compression Manager was updated to support using the alpha channel when displaying images. Alpha channels are supported only for 32-bit images. The high byte of each pixel contains the alpha channel. The alpha channel can be interpreted in one of three ways:

- straight alpha
- pre-multiplied with white
- pre-multiplied with black

QuickTime uses the alpha channel to define how an image is to be combined with the image that is already present at the location to which it will be drawn. This is similar to how QuickDraw's blend mode works. To combine an image containing an alpha channel with another image, you specify how the alpha channel should be interpreted by specifying one of the new alpha channel graphics modes defined by QuickTime.

Straight alpha means that the color components of each pixel should be combined with the corresponding background pixel based on the value contained in the alpha channel. For example, if the alpha value is 0, only the background pixel will appear. If the alpha value is 255, only the foreground pixel will appear. If the alpha value is 127, then (127/255) of the foreground pixel will be blended with (128/255) of the background pixel to create the resulting pixel, and so on.

Pre-multiplied with white means that the color components of each pixel have already been blended with a white pixel, based on their alpha channel value. Effectively, this means that the image has already been combined with a white background. To combine the image with a different background color, QuickTime must first remove the white from each pixel and then blend the image with the actual background pixels. Images are often pre-multipled with white as this reduces the appearance of jagged edges around objects.

Pre-multipled with black is the same as pre-multipled with white, except the background color that the image has been blended with is black instead of white.

Note
Although you pass these new alpha channel graphics modes to QuickTime in the same way as you would traditional QuickDraw transfer modes, these modes are not supported by QuickDraw and will cause unpredictable results if passed to QuickDraw routines. ◆

The Image Compression Manager defines the following constants for specifying alpha channel graphics modes:

```
enum {
    graphicsModeStraightAlpha       = 256,
    graphicsModePreWhiteAlpha       = 257,
    graphicsModePreBlackAlpha       = 258
    graphicsModeStraightAlphaBlend  = 260
};
```

The `graphicsModeStraightAlpha`, `graphicsModePreWhiteAlpha`, and `graphicsModePreBlackAlpha` graphics modes cause QuickTime to draw the image interpreting the alpha channel as specified. The graphics mode `graphicsModeStraightAlphaBlend` causes QuickTime to interpret the alpha channel as a straight alpha channel, but when it draws, combines the pixels together and applies the `opColor` supplied with the graphics mode to the alpha channel. This provides an easy way to combine images using both an alpha channel and a blend level. This can be useful when compositing 3D rendered images over video.

To draw a compressed image containing an alpha channel, that image must be compressed using an image-compression format that is capable of storing the alpha channel information. The Animation, Planar RGB and None compressors store alpha channel data in the “Millions of Colors +” (32-bit) mode.

You use the `MediaSetGraphicsMode` function to set a movie track to use an alpha channel graphics mode. You use the `SetDSequenceTransferMode` function to set an image sequence to use an alpha channel graphics mode.

Working With Video Fields

QuickTime 2.5 introduced support for working directly with fields of interlaced video, such as those created by some motion JPEG compressors.

Because video processing applications sometimes need to perform operations on individual fields (for example, reversing them or combining one field of a frame with a field from another frame), QuickTime now provides a method for accessing the individual fields without having to decompress them first. Previously such operations required decompressing each frame, copying the appropriate fields, and then recompressing. This was a time consuming process that could result in a loss of image quality due to the decompression and recompression of the video data.

Three new functions (`ImageFieldSequenceBegin`, `ImageFieldSequenceExtractCombine`, and `ImageFieldSequenceEnd`) allow an application to request that field operations be performed directly on the compressed data. These functions accept one or two compressed images as input and create a single compressed image on output.

The Apple Component Video and Motion JPEG compressors support image field functions in QuickTime 2.5 and later. See the description of the `ImageFieldSequenceBegin`, `ImageFieldSequenceExtractCombine`, and `ImageFieldSequenceEnd` functions for information on how to process image fields in your application. See Chapter 4, “Image Compressor Components,” for information on incorporating support for these functions in other compressors.

Packetization Information

QuickTime video compressors are increasingly being used for videoconferencing applications. Image data from a compressor is typically split into network-packet-sized pieces, transmitted through a packet-based protocol (such as UDP or DDP), and reassembled into a frame by the receiver(s). Typically, a lost packet causes an entire frame to be dropped; without all the data for a given frame, the decompressor cannot decode the image. When the loss of one packet forces others to be unusable, the loss rate is effectively multiplied by a large factor.

Some compression methods, however, such as H.261, can divide a compressed image into pieces which can be decoded independently. Some videoconferencing protocols, such as the Internet’s Real Time Protocol (RTP, RFC#1889), specify that data compressed using H.261 must be packetized into independently decodable chunks. While RTP demands this packetization information from the compressor, other protocols, such as QuickTime Conferencing’s MovieTalk protocol, can optionally use this information to effectively reduce loss rates.

QuickTime 2.5 added four new functions to support packetization: `SetCSequencePreferredPacketSize` (page 212), `SGSetPreferredPacketSize` (page 316), `SGGetPreferredPacketSize` (page 317), and `VDSetPreferredPacketSize` (page 327). In addition, the `CodecCompressParams` structure (page 236) includes a new field, `preferredPacketSizeInBytes`. See Chapter 4, “Image Compressor Components,” for information about supporting packetization in image compressor components.

For application developers, the important function is `SGSetPreferredPacketSize`, which is described in Chapter 8, “Sequence Grabber Channel Components.” The `SetCSequencePreferredPacketSize` function is described later in this chapter. For information about the `VDSetPreferredPacketSize` function, see Chapter 9, “Video Digitizer Components.”

Using the Image Compression Manager

Using Screen Buffers and Image Buffers

In QuickTime 2.1, support for screen buffers was removed. All requests for screen buffers are now converted into requests for image buffers. Applications should no longer request screen buffers.

Image Compression Manager Reference

This section describes the data types, constants, and functions added to the Image Compression Manager subsequent to QuickTime 1.6.

Data Types

Image Compression Manager Function Control Flags

This section describes the new function control flags provided by the Image Compression Manager.

```
enum {
    codecFlagDontUseNewImageBuffer      = (1L << 10),
    codecFlagInterlaceUpdate            = (1L << 11),
    codecFlagCatchUpDiff                = (1L << 12)
};
```

Flag descriptions

`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 interlacing even and odd scan lines to reduce tearing artifacts (if the decompressor supports this mode).

`codecFlagCatchUpDiff`
: Notifies the codec that the currently displayed frame is being displayed late in an attempt to “catch up” to the current frame, which only happens with compression formats that support frame differencing, You can pass this flag to any of the `DecompressSequenceFrame` calls.

Constants

This section describes the new constants provided by the Image Compression Manager.

```
/* alpha channel graphics modes */
enum {
    graphicsModeStraightAlpha       = 256,
    graphicsModePreWhiteAlpha       = 257,
    graphicsModePreBlackAlpha       = 258,
    graphicsModeStraightAlphaBlend  = 260
};

/* fieldFlags for the ImageFieldSequenceExtractCombine function */
enum {
    evenField1ToEvenFieldOut        = 1<<0,
    evenField1ToOddFieldOut         = 1<<1,
    oddField1ToEvenFieldOut         = 1<<2,
    oddField1ToOddFieldOut          = 1<<3,
    evenField2ToEvenFieldOut        = 1<<4,
    evenField2ToOddFieldOut         = 1<<5,
    oddField2ToEvenFieldOut         = 1<<6,
    oddField2ToOddFieldOut          = 1<<7
};
```

Functions

This section describes the functions added to the Image Compression Manager subsequent to QuickTime 1.6.

Working With Sequences

DecompressSequenceBeginS

Sends a sample image to a decompressor.

```
pascal OSErr DecompressSequenceBeginS (
                    ImageSequence *seqID,
                    ImageDescriptionHandle desc,
                    Ptr data,
                    long dataSize,
                    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.
`desc`	Contains a handle to the image description structure that describes the compressed image.
`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` function before you use that pointer with this parameter.
`dataSize`	Specifies the size of the `data` buffer.

`port`	Points to the graphics port for the destination image.
`gdh`	Contains a handle to the graphics device record for the destination image.
`srcRect`	Contains a pointer to a rectangle defining the portions of the image to decompress.
`matrix`	Points to a matrix structure that specifies how to transform the image during decompression.
`mode`	Specifies the transfer mode for the operation.
`mask`	Contains a handle to the clipping region in the destination coordinate system.
`flags`	Contains flags providing further control information.
`accuracy`	Specifies the accuracy desired in the decompressed image.
`codec`	Contains the compressor identifier.

DISCUSSION

The `DecompressSequenceBeginS` function, introduced in QuickTime 1.6.1, allows you to pass a compressed sample so the codec can perform preflighting before the first `DecompressSequenceFrame` call.

SetSequenceProgressProc

Installs a progress procedure for a sequence.

```
pascal OSErr SetSequenceProgressProc (
                    ImageSequence seqID,
                    ICMProgressProcRecord *progressProc);
```

`seqID`	Contains the unique sequence identifier that was returned by the `DecompressSequenceBegin` function.
`progressProc`	Points to a record containing information about the application’s progress procedure.

DISCUSSION

The `SetSequenceProgressProc` function, introduced in QuickTime 1.6.1, allows you to set a progress procedure on a compression or decompression sequence, just as earlier versions of QuickTime allowed you to set a progress procedure when compressing or decompressing a still image.

GetCSequenceMaxCompressionSize

The `GetCSequenceMaxCompressionSize` function allows your application to determine the maximum size an image will be after compression for a given compression sequence. You must have already started a compression sequence with `CompressSequenceBegin`.

```
pascal OSErr GetCSequenceMaxCompressionSize(
                    ImageSequence seqID,
                    PixMapHandle src,
                    long *size);
```

`seqID`	Contains the unique sequence identifier that was returned by the `CompressSequenceBegin` function.
`src`	Contains a handle to the source pixel map. The compressor uses only the image's size and pixel depth to determine the maximum size of the compressed image.
`size`	Contains a pointer to a field to receive the maximum size, in bytes, of the compressed image.

DISCUSSION

The `GetCSequenceMaxCompressionSize` function is similar to the `GetMaxCompressionSize` function, but operates on a compression sequence instead of requiring the application to pass the individual parameters about the source image.

DecompressSequenceFrameWhen

Queues a frame for decompression and specifies the time at which decompression will begin.

```
pascal OSErr DecompressSequenceFrameWhen (
                     ImageSequence seqID,
                     Ptr data,
                     long dataSize,
                     CodecFlags inFlags,
                     CodecFlags *outFlags,
                     ICMCompletionProcRecordPtr asyncCompletionProc,
                     const ICMFrameTimeRecord *frameTime);
```

`seqID` Contains the unique sequence identifier that was returned by the `DecompressSequenceBegin` function.

`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` function before you use that pointer with this parameter.

`dataSize` Specifies the size of the `data` buffer.

`inFlags` Contains flags providing further control information. See *Inside Macintosh: QuickTime* for information about `CodecFlags` fields. The following flags are valid 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 not 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 status flags. The decompressor updates these flags at the end of the decompression operation. See *Inside Macintosh: QuickTime* 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`
This input flag forces an error to be returned when a new image buffer would have to be allocated instead of allocating the new buffer.

`codecFlagInterlaceUpdate`
: This input flag updates the screen interlacing even and odd scan lines to reduce tearing artifacts (if the decompressor supports this mode).

`codecFlagCatchUpDiff`
: This input flag notifies the codec that the currently displayed frame is being displayed late in an attempt to “catch up” to the current frame, which only happens with compression formats that support frame differencing.

`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 *Inside Macintosh: QuickTime* 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.

`frameTime` Points to a structure that contains the frame’s time information, including the time at which the frame should be displayed, its duration, and the movie’s playback rate. This parameter can be nil, in which case the decompression operation will happen immediately.

DISCUSSION

This function, introduced with QuickTime 2.0, accepts the same parameters as the `DecompressSequenceFrame` function, with the addition of the `frameTime` and `dataSize` parameters. The `frameTime` parameter points to an `ICMFrameTime` structure, which contains the frame’s time information. The `ICMFrameTime` structure is described in Chapter 4, “Image Compressor Components.”

SPECIAL CONSIDERATIONS

If the current decompressor component does not support scheduled asynchronous decompression, the Image Compression Manager returns an error code of `codecCantWhenErr`. In this case, the application will need to reissue the request with the `frameTime` parameter set to nil. If the decompressor cannot service your request at a particular time (for example, if its queue is full), the Image Compression Manager returns an error code of `codecCantQueueErr`. The best way to determine whether a decompressor component supports this function is to call the function and test the result code. A decompressor's ability to honor the request may change based on screen depth, clipping settings, and so on.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memFullErr`	–108	Not enough memory available
`noCodecErr`	–8961	Could not find the specified decompressor
`codecSpoolErr`	–8966	Error loading or unloading data
`codecCantWhenErr`	–8974	Decompressor can't honor this request
`codecCantQueueErr`	–8975	Decompressor can't queue this frame

DecompressSequenceFrameS

Queues a frame for decompression and specifies the size of the compressed data. New applications should use `DecompressSequenceFrameWhen` (page 193).

```
pascal OSErr DecompressSequenceFrameS(
                ImageSequence seqID,
                Ptr data,
                long dataSize,
                CodecFlags inFlags,
                CodecFlags *outFlags,
                ICMCompletionProcRecordPtr asyncCompletionProc);
```

`seqID`	Contains the unique sequence identifier that was returned by the `DecompressSequenceBegin` function.

`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` function before you use that pointer with this parameter.

`dataSize` Specifies the size of the `data` buffer.

`inFlags` Contains flags providing further control information. See *Inside Macintosh: QuickTime* for information about `CodecFlags` fields. The following flags are valid 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 not 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 status flags. The decompressor updates these flags at the end of the decompression operation. See *Inside Macintosh: QuickTime* 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`
This input flag forces an error to be returned when a new image buffer would have to be allocated instead of allocating the new buffer.

`codecFlagInterlaceUpdate`
This input flag updates the screen interlacing even and odd scan lines to reduce tearing artifacts (if the decompressor supports this mode).

`codecFlagCatchUpDiff`
This input flag notifies the codec that the currently displayed frame is being displayed late in an attempt to “catch up” to the current frame, which only happens with compression formats that support frame differencing.

`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 *Inside Macintosh: QuickTime* 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.

DISCUSSION

This function, introduced in QuickTime 1.6.1, accepts the same parameters as the `DecompressSequenceFrame` function, with the addition of the `dataSize` parameter.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memFullErr`	–108	Not enough memory available
`noCodecErr`	–8961	Could not find the specified decompressor
`codecSpoolErr`	–8966	Error loading or unloading data

CDSequenceFlush

Stops a decompression sequence, aborting processing of any queued frames.

```
pascal OSErr CDSequenceFlush(ImageSequence seqID);
```

`seqID` Contains the unique sequence identifier that was returned by the `DecompressSequenceBegin` function.

DISCUSSION

This function, introduced with QuickTime 2.0, is used to tell a decompressor component to stop processing of any queued scheduled asynchronous decompression. This is useful when several frames have been queued for decompression in the future and the application needs to suspend playback of the sequence.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified

SetDSequenceTimeCode

Sets the timecode value for the frame that is about to be decompressed.

```
pascal OSErr SetDSequenceTimeCode (
                    ImageSequence seqID,
                    void TimeCodeDef *timeCodeFormat,
                    void TimeCodeTime *timeCodeTime);
```

`seqID`	Contains the unique sequence identifier that was returned by the `DecompressSequenceBegin` function.
`timeCodeFormat`	Contains a pointer to a timecode definition structure. You provide the appropriate timecode definition information for the next frame to be decompressed.
`timeCodeTime`	Contains a pointer to a timecode record structure. You provide the appropriate time value for the next frame in the current sequence.

DISCUSSION

QuickTime’s video media handler uses this function to set the timecode information for a movie. When a movie that contains timecode information starts playing, the media handler calls this function as it processes the movie’s first frame.

Note that the Image Compression Manager passes the timecode information straight through to the image decompressor component. That is, the Image Compression Manager does not make a copy of any of this timecode information. As a result, you must make sure that the data referred to by the `timeCodeFormat` and `timeCodeTime` parameters is valid until the next decompression operation completes.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memFullErr`	–108	Not enough memory available
`noCodecErr`	–8961	Could not find the specified decompressor

CDSequenceEquivalentImageDescription

Reports whether two image descriptions are the same.

```
pascal OSErr CDSequenceEquivalentImageDescription (
                        ImageSequence seqID,
                        ImageDescriptionHandle newDesc,
                        Boolean *equivalent);
```

`seqID`	Contains the unique sequence identifier that was returned by the `DecompressSequenceBegin` function.
`newDesc`	Contains a handle to the image description structure that describes the compressed image.
`equivalent`	Contains a pointer to a Boolean value. If the `ImageDescriptionHandle` provided in the `newDesc` parameter is equivalent to the image description currently in use by the image sequence, this value is set to `true`. If the `ImageDescriptionHandle` is not equivilent, and therefore a new image sequence must be created to display an image using the new image description, this value is set to `false`.

DISCUSSION

The `CDSequenceEquivalentImageDescription` function allows an application to ask whether two image descriptions are the same. If they are, the decompressor does not have to create a new image decompression sequence to display those images.

SPECIAL CONSIDERATIONS

The Image Compression Manager can only implement part of this function by itself. There are some fields in the image description that it knows are irrelevant to the decompressor. If the Image Compression Manager determines that there are differences in fields that may be significant to the codec, it calls the function `ImageCodecIsImageDescriptionEquivalent` (page 248) to ask the codec.

CDSequenceNewMemory

Requests codec-allocated memory.

```
pascal OSErr CDSequenceNewMemory (
                    ImageSequence seqID,
                    Ptr *data,
                    Size dataSize,
                    long dataUse,
                    ICMMemoryDisposedUPP memoryGoneProc,
                    void *refCon);
```

`seqID`	Contains the unique sequence identifier that was returned by the `DecompressSequenceBegin` function.
`data`	Returns a pointer to the allocated memory.
`dataSize`	Specifies the requested size of the `data` buffer.
`dataUse`	A code that indicates how the memory is to be used. For example, the memory may be used to store compressed image or mask plane data, or used as an offscreen image buffer.

If there is no benefit to storing a particular kind of data in codec memory, the codec should deny the request for the memory allocation. The defined values are for data use are:

0x0001	Memory will be used for holding compressed image data.
0x0002	Memory will be used for an offscreen image buffer.

`memoryGoneProc`
: A pointer to a function that will be called before disposing of the memory allocated by a codec. Your callback function must be in the following form:

```
pascal void (*ICMMemoryDisposedProcPtr)
                (Ptr memoryBlock, void *refcon);
```

`refCon`
: Contains a reference constant value to be passed to your `memoryGoneProc` function.

DISCUSSION

Because many newer hardware decompresson boards contain dedicated on-board memory, significant performance gains can be realized if this memory is used to store data before it is decompressed.

The decompressor can, at any time, dispose of all memory it has allocated. When memory is allocated, an `ICMMemoryDisposedProc` callback function must be provided. The decompressor calls this routine before disposing of the memory.

A callback procedure is required because memory on the hardware decompresson board may be limited. If the decompressor cannot deallocate memory as required, it is possible that an idle decompressor instance may be holding a large amount of memory, denying those resources to the currently active decompressor instance.

The decompressor memory must never be disposed at interrupt time. When the procedure is called, the memory is still available. This allows any pending reads into the block to be canceled before the block is disposed. The decompressor disposing the memory must ensure that it is not disposing a block that it is currently using (that is, the memory that contains the currently decompressing frame).

To dispose of the memory, use the `CDSequenceDisposeMemory` function.

CDSequenceDisposeMemory

Disposes of memory allocated by the codec.

```
pascal OSErr CDSequenceDisposeMemory (
                    ImageSequence seqID,
                    Ptr data);
```

`seqID`	Contains the unique sequence identifier that was returned by the `DecompressSequenceBegin` function.
`data`	Points to the previously allocated memory block.

DISCUSSION

You call this function to release memory allocated by the `CDSequenceNewMemory` function.

SPECIAL CONSIDERATIONS

Do not call the `CDSequenceDisposeMemory` function at interrupt time.

CDSequenceInvalidate

Notifies the Image Compression Manager that the destination port for the given image decompression sequence has been invalidated.

```
pascal OSErr CDSequenceInvalidate(
                    ImageSequence seqID,
                    RgnHandle invalRgn);
```

`seqID`	Contains the unique sequence identifier that was returned by the `DecompressSequenceBegin` function.
`rgn`	A handle of the region specifiying the invalid portion of the image.

DISCUSSION

You call this function to force the Image Compression Manager to redraw the screen bits on the next decompression operation.

Working With Images

PtInDSequenceData

Tests to see if an image contains data at a a given point.

```
pascal OSErr PtInDSequenceData(
                    ImageSequence seqID,
                    void *data,
                    Size dataSize,
                    Point where,
                    Boolean *hit);
```

`seqID`	Contains the unique sequence identifier that was returned by the `DecompressSequenceBegin` function.
`data`	Pointer to compressed data in the format specified by the `desc` param.
`dataSize`	Size of the compressed data referred to by the `data` param.
`where`	A QuickDraw Point. 0,0 based at the top-left corner of the image.
`hit`	A pointer to a field to receive the Boolean indicating whether or not the image contained data at the specified point. The Boolean will be set to `true` if the point specified by the `where` parameter is contained within the compressed image data specified by the `data` param.

DISCUSSION

The `PtInDSequenceData` function allows the application to perform hit testing on compressed data. The `hit` parameter will be set to true if the compressed data contains data at the point specified by the `where` parameter. The `hit` parameter will be set to false if the specified point falls within a blank portion of the image.

Working With Data Sources

CDSequenceNewDataSource

Creates a new data source.

```
pascal OSErr CDSequenceNewDataSource (
                    ImageSequence seqID,
                    ImageSequenceDataSource *sourceID,
                    OSType sourceType,
                    long sourceInputNumber,
                    Handle dataDescription,
                    void *transferProc,
                    void *refCon);
```

`seqID` The unique sequence identifier that was returned by the `DecompressSequenceBegin` function.

`sourceID` Returns the new data source identifier.

`sourceType` A four-character code describing how the input will be used. This code is usually derived from the information returned by the codec. For example, if a mask plane was passed, this field might contain 'mask'.

`sourceInputNumber`
More than one instance of a given source type may exist. The first occurrence should have a source input number of 1, the second a source input number of 2, and so on.

`dataDescription`
A handle to a data structure describing the input data. For compressed image data, this is just an `ImageDescriptionHandle`.

`transferProc` A routine that allows the application to transform the type of the input data to the kind of data preferred by the codec. The client of the codec passes the source data in the form most convenient for it. If the codec needs the data in another form, it can negotiate with the client or directly with the Image Compression Manager to obtain the required data format.

`refCon`	Contains a reference constant value to be passed to the transfer procedure.

DISCUSSION

This function returns a `sourceID` parameter which must be passed to all other functions that reference the source. All data sources are automatically disposed when the sequence they are associated with is disposed.

CDSequenceDisposeDataSource

Disposes of a data source.

```
pascal OSErr CDSequenceDisposeDataSource (
                    ImageSequenceDataSource sourceID);
```

`sourceID`	The data source identifier that was returned by the `CDSequenceNewDataSource` function.

DISCUSSION

You use this function to dispose of a data source created by the `CDSequenceNewDataSource` function. All data sources are automatically disposed when the sequence they are associated with is disposed.

CDSequenceSetSourceData

Sets data in a new frame to a specific data source.

```
pascal OSErr CDSequenceSetSourceData (
                    ImageSequenceDataSource sourceID,
                    void *data,
                    long dataSize);
```

`sourceID`	Contains the source identifier of the data source.
`data`	Points to the 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` function before you use that pointer with this parameter.
`dataSize`	Specifies the size of the `data` buffer.

DISCUSSION

The `CDSequenceSetSourceData` function is called to set data in a new frame to a specific source. For example, as a new frame of compressed data arrives at a source, `CDSequenceSetSourceData` will be called.

CDSequenceChangedSourceData

Notifies the compressor that the image source data has changed.

```
pascal OSErr CDSequenceChangedSourceData (
                    ImageSequenceDataSource sourceID);
```

`sourceID`	Contains the source identifier of the data source.

DISCUSSION

Use the new `CDSequenceSetChangedSourceData` function to indicate that the image has changed but the data pointer to that image has not changed. For example, if the data pointer points to the base address of a PixMap, the image in the PixMap can change, but the data pointer remains constant.

Working With Image Description Records

AddImageDescriptionExtension

Adds an extension to an `ImageDescriptionHandle`.

```
pascal OSErr AddImageDescriptionExtension(
                        ImageDescriptionHandle desc,
                        Handle extension,
                        long idType);
```

`desc`	The handle of the ImageDescription to add the extension to.
`extension`	The handle containing the extension data.
`idType`	A four-byte signature indentifying the type of data being added to the ImageDescription.

DISCUSSION

This function allows the application to add custom data to an ImageDescriptionHandle. This data could be specific to the compressor component referenced by the image description.

SPECIAL CONSIDERATIONS

The Image Compression Manager makes a copy of the data referred to by the `extension` parameter. Thus, your application should dispose its copy of the data when it is no longer needed.

GetNextImageDescriptionExtensionType

Adds an extension to an `ImageDescriptionHandle`.

```
pascal OSErr GetNextImageDescriptionExtensionType(
                    ImageDescriptionHandle desc,
                    long *idType);
```

`desc`	The `ImageDescriptionHandle`.
`idType`	A pointer to a field that, on entry, contains the starting point for the search. On return, will contain the next extension type found in the `ImageDescriptionHandle`.

DISCUSSION

This function allows the application to search for all types of extensions in an `ImageDescriptionHandle`. The `idType` field should be set to 0 to start the search. When no more extension types can be found, this field will be set to 0.

CountImageDescriptionExtensionType

Counts the number of extensions of a given type in an `ImageDescriptionHandle`.

```
pascal OSErr CountImageDescriptionExtensionType(
                    ImageDescriptionHandle desc,
                    long idType,
                    long *count);
```

`desc`	The `ImageDescriptionHandle`.
`idType`	A four-byte signature indentifying the type of extension data.
`count`	A pointer to a field to receive the number of extensions of the specified type.

DISCUSSION

This function, when used with the "`GetNextImageDescriptionExtensionType`" call, allows the application to determine the total set of extensions present in the `ImageDescriptionHandle`.

GetImageDescriptionExtension

Returns a new handle with the data from a specified image description extension.

```
pascal OSErr GetImageDescriptionExtension(
                    ImageDescriptionHandle desc,
                    Handle *extension,
                    long idType,
                    long index);
```

`desc`	The `ImageDescriptionHandle`.
`extension`	A pointer to a field to receive a new handle with the extension data.
`idType`	The type of extension to receive.
`index`	The index (from 1 to the count as returned by "`CountImageDescriptionExtensionType`") of the extension to receive.

DISCUSSION

This function allows the application to get a copy of a specified image description extension.

SPECIAL CONSIDERATIONS

The Image Compression Manager allocates a new handle and passes it back in the `extension` parameter. Your application should dispose of the handle when it is no longer needed.

RemoveImageDescriptionExtension

Removes a specified extension from an `ImageDescriptionHandle`.

```
pascal OSErr RemoveImageDescriptionExtension(
                 ImageDescriptionHandle desc,
                 long idType,
                 long index);
```

`desc`	The `ImageDescriptionHandle`.
`idType`	The type of extension to receive.
`index`	The index (from 1 to the count as returned by "`CountImageDescriptionExtensionType`") of the extension to receive.

DISCUSSION

This function allows the application to remove a specified extension from an `ImageDescriptionHandle`. Note that any extensions that are present in the `ImageDescriptionHandle` after the deleted extension will have their index numbers renumbered.

Changing Sequence Compression Parameters

SetCSequencePreferredPacketSize

Sets the preferred packet size for a sequence.

```
pascal OSErr SetCSequencePreferredPacketSize (
                 ImageSequence seqID,
                 long preferredPacketSizeInBytes);
```

`seqID`	The sequence identifier.
`preferredPacketSizeInBytes`	The preferred packet size in bytes.

DISCUSSION

This function was added in QuickTime 2.5 to support video conferencing applications by making each transmitted packet an independently decodable chunk of data.

Controlling Hardware Scaling

QuickTime 1.6.1 added three functions that allow applications to zoom a monitor (`GDHasScale`, `GDGetScale`, and `GDSetScale`). These three functions are considered low-level calls (comparable to `SetEntries`) that you should use only when playing back QuickTime movies in a controlled environment with no user interaction. Also, because this capability is not present on all computers, applications should not depend on its availability.

These new functions provide a standard way for you to access the resizing abilities of a user's monitor for playback. Effectively, this allows you to have full screen Cinepak playback on low-end Macintosh computers.

GDHasScale

Returns the closest possible scaling that a particular screen device can be set to in a given pixel depth.

```
pascal OSErr GDHasScale (
                    GDHandle gdh,
                    short depth,
                    Fixed *scale);
```

`gdh`	Contains a handle to a screen graphics device.
`depth`	Specifies the pixel depth of the screen device.
`scale`	Points to a fixed point scale value. On input, this field should be set to the desired scale value. On output, this field will contain the closest scale available for the given depth. A scale of 0x10000 indicates normal size, 0x20000 indicates double size, and so on.

DISCUSSION

The `GDHasScale` function returns scaling information for a particular GDevice for a requested depth. This function allows you to query a GDevice without actually changing it. For example, if you specify 0x20000 but the GDevice does not support it, `GDHasScale` returns with noErr and a scale of 0x10000. Because this function checks for a supported depth, your requested depth must be supported by the GDevice. `GDHasScale` references the video driver through the graphics device structure.

RESULT CODES

`CDepthErr`	–157	The requested depth is not supported
`cDevErr`	–155	Not a screen device
`controlErr`	–17	Video driver cannot respond to this call

GDGetScale

Returns the current scale of the given screen graphics device.

```
pascal OSErr GDGetScale (
                    GDHandle gdh,
                    Fixed *scale,
                    short *flags);
```

`gdh`	Contains a handle to a screen graphics device.
`scale`	Points to a fixed point field to hold the scale result.
`flags`	Points to a short integer. It returns the status parameter flags for the video driver. For now, 0 is always returned in this field.

RESULT CODES

`cDevErr`	–155	Not a screen device
`controlErr`	–17	Video driver cannot respond to this call

GDSetScale

Sets a screen graphics device to a new scale.

```
pascal OSErr GDSetScale (
                    GDHandle gdh,
                    Fixed scale,
                    short flags);
```

`gdh`	Contains a handle to a screen graphics device.
`scale`	A fixed point scale value.
`flags`	Points to a short integer. It returns the status parameter flags for the video driver. For now, 0 is always returned in this field.

RESULT CODES

`cDevErr`	–155	Not a screen device
`controlErr`	–17	Video driver cannot respond to this call

Working With Video Fields

QuickTime 2.5 introduced three functions for working with compressed fields of video data. The `ImageFieldSequenceBegin` function initiates an image field sequence operation; the `ImageFieldSequenceExtractCombine` function performs the desired operations; and the `ImageFieldSequenceEnd` function terminates the operation.

ImageFieldSequenceBegin

Initiates an image field sequence operation and specifies the input and output data format.

```
pascal OSErr ImageFieldSequenceBegin (
                        ImageFieldSequence *ifs,
                        ImageDescriptionHandle desc1,
                        ImageDescriptionHandle desc2,
                        ImageDescriptionHandle descOut);
```

`ifs`	On return, contains the unique sequence identifier assigned to the sequence.
`desc1`	An image description structure describing the format and characteristics of the data to be passed to the `ImageFieldSequenceExtractCombine` function through the `data1` parameter.
`desc2`	An image description structure describing the format and characteristics of the data to be passed to the `ImageFieldSequenceExtractCombine` function through the `data2` parameter. Set to `nil` if the requested operation uses only one input frame.
`descOut`	Specifies the desired format of the resulting frames. Typically this is the same format specified by the `desc1` and `desc2` parameters.

DISCUSSION

You use the `ImageFieldSequenceBegin` function to set up an image field sequence operation and specify the input and output data format.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memFullErr`	–108	Not enough memory available

ImageFieldSequenceExtractCombine

Performs field operations on video data.

```
pascal OSErr ImageFieldSequenceExtractCombine (
                    ImageFieldSequence ifs,
                    long fieldFlags,
                    void *data1,
                    long dataSize1,
                    void *data2,
                    long dataSize2,
                    void *outputData,
                    long *outDataSize);
```

`ifs` The unique sequence identifier that was returned by the `ImageFieldSequenceBegin` function.

`fieldFlags` Flags specifying the operation to be performed. A correctly formed request will specify two input fields, mapping one to the odd output field and the other to the even output field. The following flags are defined:

`evenField1ToEvenFieldOut`
Maps the even field specified by the `data1` parameter to the even output field.

`evenField1ToOddFieldOut`
Maps the even field specified by the `data1` parameter to the odd output field.

`oddField1ToEvenFieldOut`
Maps the odd field specified by the `data1` parameter to the even output field.

`oddField1ToOddFieldOut`
Maps the odd field specified by the `data1` parameter to the odd output field.

`evenField2ToEvenFieldOut`
Maps the even field specified by the `data2` parameter to the even output field.

`evenField2ToOddFieldOut`
: Maps the even field specified by the `data2` parameter to the odd output field.

`oddField2ToEvenFieldOut`
: Maps the odd field specified by the `data2` parameter to the even output field.

`oddField2ToOddFieldOut`
: Maps the odd field specified by the `data2` parameter to the odd output field.

`data1`	A pointer to a buffer containing the data of input field one.
`dataSize1`	Specifies the size of the `data1` buffer.
`data2`	A pointer to a buffer containing the data of input field two. Set to `nil` if the requested operation uses only one input frame.
`dataSize2`	Specifies the size of the `data2` buffer. Set to 0 if the requested operation uses only one input frame.
`outputData`	A pointer to a buffer to receive the resulting frame. Use the `GetMaxCompressionSize` function to determine the amount of memory to allocate for this buffer.
`outDataSize`	On output this parameter returns the actual size of the data.

DISCUSSION

This function was introduced in QuickTime 2.5 and provides a method for working directly with fields of interlaced video. You can use the `ImageFieldSequenceExtractCombine` function to change the field dominance of an image by reversing the two fields, or to create or remove the effects of the 3:2 pulldown commonly performed when transferring film to NTSC videotape.

Because this function operates directly on the compressed video data, it is faster than working with decompressed images. It also has the added benefit of eliminating any image quality degradation that might result from lossy codecs.

The `ImageFieldSequenceExtractCombine` function accepts one or two compressed images as input and creates a single compressed image on output. You specify the operation to be performed using the `fieldFlags` parameter. The function returns the `codecUnimpErr` result code if there is no codec present in the system that can perform the requested operation.

The Apple Component Video (YUV) and Motion JPEG codecs currently support this function. See Chapter 4, “Image Compressor Components,” for information on incorporating support for this function in your codec.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memFullErr`	–108	Not enough memory available
`codecUnimpErr`	–8962	Feature not implemented by this compressor

ImageFieldSequenceEnd

Ends an image field sequence operation.

```
pascal OSErr ImageFieldSequenceEnd (ImageFieldSequence ifs);
```

`ifs`	The unique sequence identifier that was returned by the `ImageFieldSequenceBegin` function.

DISCUSSION

You must call this function to terminate an image field sequence operation.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified

Image Transcoding Functions

ImageTranscodeSequenceBegin

Initiates an image transcoder sequence operation.

```
pascal OSErr ImageTranscodeSequenceBegin (
                    ImageTranscodeSequence *its,
                    ImageDescriptionHandle srcDesc,
                    OSType destType,
                    ImageDescriptionHandle *dstDesc
                    void *data,
                    long dataSize);
```

`its`	The image transcoder sequence identifier. If the operation fails, the value pointed to by `its` is set to `nil`.
`srcDesc`	The image description for the source compressed image data.
`destType`	The desired compression format into which to transcode the source data.
`dstDesc`	Returns an image description for the data which will be generated by the image transcoding sequence.
`data`	Pointer to first frame of compressed data to transcode. Set to `nil` of not available.
`dataSize`	Size of the compressed data, in bytes. Set to zero if no data is provided.

DISCUSSION

This function begins an image transcoder sequence operation and returns the sequence identifier in the `its` parameter. The caller is responsible for disposing of the image description that is returned in the `dstDesc` parameter. If no transcoder is available to perform the requested transcoding operation, a `handlerNotFound` error is returned.

ImageTranscodeFrame

Transcodes a frame of image data.

```
pascal OSErr ImageTranscodeFrame (
                    ImageTranscodeSequence its,
                    void *srcData,
                    long srcDataSize,
                    void **dstData,
                    long *dstDataSize);
```

`its`	Specifies the image transcoder sequence to use to perform the transcoding operation.
`srcData`	Contains a pointer to the source data to transcode.
`srcDataSize`	Indicates the size of the compressed source image data in bytes.
`dstData`	Returns a pointer to the transcoded image data.
`dstDataSize`	Returns the size of the transcoded image data.

DISCUSSION

After creating the image transcoder sequence using `ImageTranscodeSequenceBegin`, you use the `ImageTranscodeFrame` function to transcode a frame of image data. The caller is responsible for disposing of the transcoded data using the `ImageTranscodeDisposeFrameData` function.

ImageTranscodeDisposeFrameData

Disposes transcoded image data.

```
pascal OSErr ImageTranscodeDisposeFrameData(
                    ImageTranscodeSequence its,
                    void *dstData)
```

`its`	Specifies the image transcoder sequence that was used to generate the transcoded data.

`dstData` Contains a pointer to the transcoded image data generated by the `ImageTranscodeFrame` function.

DISCUSSION

When the transcoded image data returned by `ImageTranscodeFrame` is no longer needed, use the `ImageTranscodeDisposeFrameData` function to dispose of the data. Only the image transcoder that generated the data can properly dispose of it.

ImageTranscodeSequenceEnd

Ends an image transcoder sequence operation.

The only parameter to `ImageTranscodeSequenceEnd` is the identifier of the image transcoder sequence to dispose. It is safe to pass a value of 0 to this routine.

```
pascal OSErr ImageTranscodeSequenceEnd (ImageTranscodeSequence its)
```

`its` The identifier of the image transcoder sequence to dispose. It is safe to pass a value of 0 in this parameter.

DISCUSSION

You must call this function to terminate an image transcoder sequence operation and dispose of the sequence.

C H A P T E R 4

Image Compressor Components

This chapter discusses new features and changes to image compressor components as documented in Chapter 4 of *Inside Macintosh: QuickTime Components*.

New Features of Image Compressor Components

Asynchronous Decompression

In QuickTime 2.0 the Image Compression Manager (ICM) was enhanced to support scheduled asynchronous decompression operations. By calling the Image Compression Manager function `DecompressSequenceFrameWhen` (page 193), applications can schedule decompression requests in advance. This allows decompressor components that support this functionality to provide reliable playback performance under a wider range of conditions.

The Apple Cinepak, Video, Animation, Component Video, and Graphics decompressors provided in QuickTime versions 2.0 and later support scheduled asynchronous decompression to 8-, 16-, and 32-bit destinations (the Cinepak decompressor also supports 4-bit grayscale destinations). QuickTime 2.5 added asynchronous decompression support to the JPEG and None decompressor components on PowerPC systems (with the QuickTime PowerPlug extension installed).

If you want to support this functionality, you must modify your decompressor component in the following ways:

- Report your component's new capabilities in its compressor capability structure by setting the `codecCanAsyncWhen` and `codecCanAsync` flags.

- Modify your component's `ImageCodecBandDecompress` (page 245) function to accept scheduled asynchronous decompression requests and process them correctly.
- Implement the new function `ImageCodecFlush` (page 246); this function allows the Image Compression Manager to instruct you to empty your input queue.
- Optionally, implement logic to manage the shielding of the cursor during decompression operations.

All of these changes are discussed in detail in "Image Compressor Components Reference" beginning on page 232.

Hardware Cursors

The Image Compression Manager supports hardware cursors introduced in PCI-based Macintosh computers, which eliminates cursor flicker. For all software codecs, this support requires no changes.

For codecs that manage the cursor themselves, QuickTime 2.1 provided a flag, `codecCompletionDontUnshield`, for use when calling the `ICMDecompressComplete` (page 259) function. Use this flag to prevent the Image Compression Manager from unshielding the cursor when `ICMDecompressComplete` is called.

Timecode Support

As discussed in Chapter 3, "Image Compression Manager," timecode support was added to the Image Compression Manager in QuickTime 2.0. QuickTime 2.0 and above has continued to be enhanced to provide timecode information to decompressor components when movies are played. This feature is provided for hardware systems that may want to use timecode information.

To support timecode in your image compressor component, your codec must support the function `ImageCodecSetTimeCode` (page 247), which allows the Image Compression Manager to set the timecode value for the next frame to be decompressed. For more information about timecode tracks and the timecode media handler, see Chapter 1, "Movie Toolbox."

Working With Video Fields

The functionality of the `ImageFieldSequenceExtractCombine` function described in Chapter 3, "Image Compression Manager," is performed by individual image codecs. This is because the way in which fields are stored is different for every compression format. A new codec component function, `ImageCodecExtractAndCombineFields` (page 242), has been defined for this purpose. Apple encourages developers of codecs to incorporate this function, if their compressed data format is capable of separately storing both fields of a video frame.

Accelerated Video Support

QuickTime 2.5 contained new support for developers of codecs to accelerate certain image decompression operations. These features will most likely be used by developers of video hardware boards that provide special acceleration features, such as arbitrary scaling or color space conversion.

Prior to QuickTime 2.5, if a codec could not decompress an image directly to the screen, the ICM would prepare an offscreen buffer for the codec, then use the None codec to transfer the image from the offscreen buffer to the screen. With QuickTime 2.5, if a codec cannot decompress directly to the screen it has the option of specifying that it can decompress to one or more types of non-RGB pixel spaces, specified as an `OSType` (e.g., `'yuvs'`). The ICM then attempts to find a decompressor component of that type (a transfer codec) that can transfer the image to the screen. Since the ICM does not define non-RGB pixel types, the transfer codec must support additional calls to set up the offscreen. If a transfer codec cannot be found that supports the specified non-RGB pixel types, the ICM uses the None codec with an RGB offscreen buffer.

The real speed benefit comes from the fact that since the transfer codec defines the offscreen buffer, it can place the buffer in on-board memory, or even point to an overlay plane so that the offscreen image really is on the screen. In this case, the additional step of transferring the bits from offscreen memory on to the screen is avoided.

For an image decompressor component to indicate that it can decompress to non-RGB pixel types, it should, in the `ImageCodecPreDecompress` call, fill in the `wantedDestinationPixelTypes` field with a handle to a zero-terminated list of pixel types that it can decompress to. The ICM immediately makes a copy of the handle. Cinepak, for example, returns a 12-byte handle containing `yuvs`, `yuvu`, and $00000000. Since `ImageCodecPreDecompress` can be called often, it is suggested that codecs allocate this handle when their component is opened and simply fill in the `wantedDestinationPixelTypes` field with this handle during `ImageCodecPreDecompress`. Components that use this method should be sure to dispose the handle at close.

Apple's Cinepak decompressor supports decompressing to `'yuvs'` and `'yuvu'` pixel types. Type `'yuvs'` is a YUV format with u and v components signed (center point at $00), while `'yuvu'` has the u and v component centered at $80. The YUV format used by QuickTime is described in "YUV" (page 988).

As an example, suppose XYZ Co. had a video board that had a YUV overlay plane capable of doing arbitrary scaling. The overlay plane takes data in the same format as Cinepak's `'yuvs'` format. In this case, XYZ would make a component of type `'imdc'` and subtype `'yuvs'`.

The `CDPreDecompress` call would set the `codecCanScale`, `codecHasVolatileBuffer`, and `codecImageBufferIsOnScreen` bits in the `capabilities->flags` field. The `codecImageBufferIsOnScreen` bit is necessary to inform the ICM that the codec is a direct screen transfer codec. A direct screen transfer codec is one that sets up an offscreen buffer that is actually onscreen (such as an overlay plane). Not setting this bit correctly can cause unpredictable results.

The real work of the codec takes place in the `CDCodecNewImageBufferMemory` call. This is where the codec is instructed to prepare the non-RGB pixel buffer. The codec must fill in the `baseAddr` and `rowBytes` fields of the `dstPixMap` structure in the `CodecDecompressParams`. The ICM then passes these values to the original codec (e.g., Cinepak) to decompress into.

The codec must also implement `CDDisposeMemory` to balance `CDCodecNewImageBufferMemory`.

Since Cinepak then decompresses into the card's overlay plane, `CDBandDecompress` needs to do nothing aside from calling `ICMDecompressComplete`.

```
pascal ComponentResult
CDPreDecompress(Handle storage,
        CodecDecompressParams *p)
{
    CodecCapabilities   *capabilities = p->capabilities;

    // only allow 16 bpp source
    if ((**p->imageDescription).depth != 16)
        return codecConditionErr;

    /* we only support 16 bits per pixel dest */
    if (p->dstPixMap.pixelSize != 16)
        return codecConditionErr;
    capabilities->wantedPixelSize = p->dstPixMap.pixelSize;

    capabilities->bandInc = capabilities->bandMin =
                (*p->imageDescription)->height;

    capabilities->extendWidth = 0;
    capabilities->extendHeight = 0;

    capabilities->flags =
            codecCanScale | codecImageBufferIsOnScreen |
            codecHasVolatileBuffer;

    return noErr;
}

pascal ComponentResult

CDBandDecompress(Handle storage,
        CodecDecompressParams *p)
{
    ICMDecompressComplete(p->sequenceID, noErr,
                codecCompletionSource | codecCompletionDest,
                &p->completionProcRecord);

    return noErr;
}

pascal ComponentResult
CDCodecNewImageBufferMemory(Handle storage,
        CodecDecompressParams *p, long flags,
        ICMMemoryDisposedUPP memoryGoneProc,
        void *refCon)
{
    OSErr err = noErr;
    long offsetH, offsetV;
    Ptr baseAddr;
    long rowBytes;
    // call predecompress to check to make sure we can handle
    // this destination
    err = CDPreDecompress(storage, p);
    if (err) goto bail;

    // set video board registers with the scale
    XYZVideoSetScale(p->matrix);

    // calculate a base address to write to
    offsetH = (p->dstRect.left - p->dstPixMap.bounds.left);
    offsetV = (p->dstRect.top - p->dstPixMap.bounds.top);
    XYZVideoGetBaseAddress(p->dstPixMap, offsetH, offsetV,
                &baseAddr, &rowBytes);

    p->dstPixMap.baseAddr = baseAddr;
    p->dstPixMap.rowBytes = rowBytes;
    p->capabilities->flags = codecImageBufferIsOnScreen;

bail:
    return err;
}

pascal ComponentResult
CDDisposeMemory(Handle storage, Ptr  data)
{
    return noErr;
}
```

Some video hardware boards that use an overlay plane require that the image area on screen be flooded with a particular RGB value or alpha-channel in order to have the overlay buffer "show through" at that location. Codecs that require this support should set the `screenFloodMethod` and `screenFloodValue` fields of the `CodecDecompressParams` record during `ImageCodecPreDecompress`. The ICM then manages the flooding of the screen buffer. This method is more reliable than having the codec attempt to flood the screen itself, and will ensure compatibility with future versions of QuickTime.

Packetization Information

QuickTime 2.5 included new functions to support packetizing compressed data streams, primarily for video conferencing applications. These functions are discussed in Chapter 3, “Image Compression Manager,” and other chapters of this book. In addition, a new field (`preferredPacketSizeInBytes`) has been added to the compression parameters structure (page 236). Codec developers need only use this field.

Packet information is appended, word-aligned, to the end of video data. It is a variable-length array of 4-byte integers, each representing the offset in bits of the end of a packet, followed by another integer containing the number of packet hints, and finally a four-byte identifier indicating the type of appended data:

```
[boundary #1][boundary #2]...[boundary #N][N]['pkts']
```

Packets are given in bits, because some types of compressed image data (such as H.261) are cut up on bit-boundaries rather than byte-boundaries.

```
// given:   image data, length, and a packet number

// returns: a pointer to the start of the packet and a packet size, plus
// information about leading and trailing bits

char* GetNextPacket(char* data, int len, int packet, long* packet_size,
    char* leading_bits, char* trailing_bits)
{
    long *lp, packets;
    lp = (long*) data;  // 'data' must be word-aligned
    lp += len/4 - 1;

    if (*lp != 'pkts')
        return nil;

    packets = *lp[-1];           // negative indexing is good for you
    if (packet >= packets)
        return nil;              // out of bounds

    lp -= packets;      // now 0-indexing into the packet array will work

    if (packet == 0)
    {
        *packet_size = (lp[0] + 7)/8;   // count the bits
        *leading_bits = 0;
        *trailing_bits = lp[0] % 8;
        return data;                    // in case of 0-length packet
    }
    else
    {
        *packet_size = ( lp[pktnum] - lp[pktnum-1] + 7) / 8;
        *leading_bits = lp[packet-1] % 8 ? 8 - lp[packet-1] % 8 : 0;
        *trailing_bits = lp[packet] % 8;
        return data + lp[packet-1] / 8;
    }
}
```

Note that this can be used for further extensions with the addition of further appended formats. The last two words are always a number of words and an extension identifier.

New Image Compressor Components

QuickTime 3 includes a compressor component and a decompressor component for DV video. These components are described in the following sections.

DV Image Compressor Component

The DV image compressor component makes it possible to compress QuickTime video data into DV format. It is invoked automatically by the Image Compression Manager when an application requests output of type `kDVCNTSCCodecType` for NTSC DV data or `kDVCPALCodecType` for PAL DV data.

When creating NTSC video, the DV image compressor component generates 720 X 480 frames. When creating PAL video, it generates 720 X 576 frames.

Note
Many DV devices use IEEE 1394 (FireWire) serial connections for input/output operations. QuickTime supports compression and decompression of DV data, but it does not include support for FireWire communication. You need additional software to communicate with DV devices. ◆

DV Image Decompressor Component

The DV image decompressor component makes it possible to decompress DV video data. It is invoked automatically by the Image Compression Manager when an application specifies input of type `kDVCNTSCCodecType` for NTSC DV data or `kDVCPALCodecType` for PAL DV data.

There are two quality modes for DV decompression:

- In the low-quality mode, which is the default, the DV image decompressor component generates a 1/4-screen image. When operating in this mode, the component uses approximately 25% of the video data in the DV stream and correspondingly fewer system resources.
- In the high-quality mode, the component processes all of the video data in the DV stream. Applications can specify the high-quality modem by calling the `SetMediaPlayHints` function with the `hintsHighQuality` flag set.

When a computer includes a video display adapter that performs YUV decompression in hardware, the DV image decompressor can use a YUV decompressor component written to use the hardware decompression capabilities in place of the software YUV decompressor in QuickTime, resulting in even higher performance.

Specifying the Size of an Image Buffer

You can now specify the size of the image buffer used by your image compressor or decompressor component. When your component calls the `CDPreDecompress` or `CDPreCompress` function, you can specify the size of the buffer as follows:

1. In the `CodecDecompressParams` or `CodecCompressParams` record, set the `codecWantsSpecialScaling` flag in the `flags` field of the `CodecCapabilities` record.
2. Provide values for the `requestedBufferWidth` and `requestedBufferHeight` fields in the `CodecDecompressParams` or `CodecCompressParams` record.

This is illustrated in Listing 4-1.

Listing 4-1 Specifying the size of an image buffer for a codec

```
p->capabilities->flags |= codecWantsSpecialScaling;
p->requestedBufferWidth = 720;
p->requestedBufferHeight = 480;
```

Image Compressor Components Reference

Data Types

The Frame Time Structure

The `ICMFrameTime` structure is defined as follows:

```
struct ICMFrameTimeRecord {
    Int64Bit                value;          /* time to display frame */
    long                    scale;          /* time scale */
    void                    *base;          /* reference to time base */
    long                    duration;       /* display duration */
    Fixed                   rate;           /* movie's playback rate */
};
```

Field descriptions

`value`	Specifies the time at which the frame is to be displayed.
`scale`	Indicates the units for the frame’s display time.
`base`	Refers to the time base.
`duration`	Specifies the duration for which the frame is to be displayed. This must be in the same units as specified by the `scale` field.
`rate`	Indicates the time base’s effective rate.

The Decompression Data Source Structure

The `CDSequenceDataSource` structure contains a linked list of all data sources for a decompression sequence. Because each data source contains a link to the next data source, a codec can access all data sources from this field. The `CDSequenceDataSource` structure is defined as follows:

```
struct CDSequenceDataSource {
    long                        recordSize;
    void *                      next;
    ImageSequence               seqID;
    ImageSequenceDataSource     sourceID;
    OSType                      sourceType;
    long                        sourceInputNumber;
    void *                      dataPtr;
    Handle                      dataDescription;
    long                        changeSeed;
    ICMConvertDataFormatUPP     transferProc;
    void *                      transferRefcon;
    long                        dataSize;
};
typedef struct CDSequenceDataSource CDSequenceDataSource;
typedef CDSequenceDataSource *CDSequenceDataSourcePtr;
```

Field descriptions

`recordSize`	Specifies the size of the record.
`next`	Contains a pointer to the next source entry. If it is `nil`, there are no more entries.
`seqID`	Specifies the image sequence that this source is associated with.
`sourceID`	Specifies the source reference identifying this source.
`sourceType`	A four-character code describing how the input will be used. This value is passed to this parameter by the `CDSequenceNewDataSource` function when the source is created.
`sourceInputNumber`	This value is passed to this parameter by the `CDSequenceNewDataSource` function when the source is created.
`dataPtr`	Contains a pointer to the actual source data.

`dataDescription`
: Contains a handle to a data structure describing the data format. This is often an image description handle.

`changeSeed`
: Contains an integer that is incremented each time the `dataPtr` field changes or that data that the `dataPtr` field points to changes. By remembering the value of this field and comparing to the value the next time the decompressor or compressor component is called, the component can determine if new data is present.

`transferProc`
: Reserved

`transferRefcon`
: Reserved

`datasize`
: Specifies the size of the `data` pointer to the `dataPtr` field.

The Compressor Capability Structure

These new compressor capability flags have been added to the compressor capability structure. Your component sets these flags in the `flags` field of the `CodecCapabilities` structure:

```
enum {
    codecCanAsyncWhen               = 1L << 16,
    codecCanShieldCursor            = 1L << 17,
    codecCanManagePrevBuffer        = 1L << 18,
    codecHasVolatileBuffer          = 1L << 19,
    codecImageBufferIsOnScreen      = 1L << 21,
    codecWantsDestinationPixels     = 1L << 22,
    codecWantsSpecialScaling        = 1L << 23
};
```

Flag descriptions

`codecCanAsyncWhen`
: Indicates whether your decompressor component supports scheduled asynchronous decompression. Set this flag to 1 if your component can support the scheduling functionality of the `CDBandDecompress` function. Note that you must also set the `codecCanAsync` flag to 1.

`codecCanShieldCursor`

Indicates whether your decompressor component will manage the shielding of the cursor during decompression. If your component can manage the cursor's display, set this flag to 1. Your component can use the Image Compression Manager's `ICMShieldSequenceCursor` function to shield the cursor. The cursor is automatically unsheilded when you call `ICMDecompressComplete` (page 259).

Otherwise, set this flag to 0—the Image Compression Manager then manages the cursor for you.

It is highly recommended that you support this capability if your decompressor supports asynchronous operation or the cursor may remain shielded for unacceptably long periods of time.

`codecCanManagePrevBuffer`

Indicates that your compressor component is capable of allocating and managing the `prevPixMap` used in temporal compression. If this flag is set, then your compressor must determine when to update the `prevPixMap` during compression sequences. Codecs setting this flag should also set `codecCanCopyPrev`.

`codecHasVolatileBuffer`

Some hardware decompressors don't actually draw the decompressed pixels into the frame buffer as requested by QuickTime. Instead, they have a second frame buffer that floats or overlays above the main frame buffer. The image is decompressed into this secondary frame buffer instead. To the user, the effect is the same because the video hardware merges the two frame buffers together. When the window that contains the image is moved to another location in the same screen buffer, the Window Manager uses `CopyBits` to transfer the window's pixels from the old location to the new location. Unfortunately, because the Window Manager is unaware of the presence of the secondary frame buffer, it cannot move the image it is displaying.

By setting the `codecHasVolatileBuffer` flag to 1, the decompressor component informs QuickTime that it uses a secondary frame buffer. When the Window Manager moves a window, QuickTime forces a redraw of the contents of the window so that the secondary frame buffer can be repositioned and/or updated as necessary.

`codecImageBufferIsOnScreen`
By setting the `codecImageBufferIsOnScreen` flag to 1, the decompressor component informs QuickTime that it is a direct screen transfer codec. Codecs that use the `CDCodecNewImageBufferMemory` call to create an offscreen buffer that is really onscreen would set this flag. See "Accelerated Video Support" (page 225) for more information on this flag.

`codecWantsSpecialScaling`
Specifies to use an image buffer whose size is determined by the `requestedBufferWidth` and `requestedBufferHeight` fields in the decompression parameters structure (`CodecDecompressParams`) or compression parameters structure (`CodecCompressParams`).

The Compression Parameters Structure

These new fields have been added to the compression parameters structure originally documented in *Inside Macintosh: QuickTime Components* .

```
struct CodecCompressParams
{
...
    /* The following fields are defined in QuickTime 2.1 or later */
    UInt16                      majorSourceChangeSeed;
    UInt16                      minorSourceChangeSeed;
    CDSequenceDataSourcePtr     sourceData;

    /* The following field is defined in QuickTime 2.5 or later */
    long                        preferredPacketSizeInBytes;

    /* The following fields are defined in QuickTime 3 or later */
    long                        requestedBufferWidth;
    long                        requestedBufferHeight;
}
```

Field description

`majorSourceChangeSeed`	Contains an integer value that is incremented each time a data source is added or removed. This provides a fast way for a codec to know when it needs to redetermine which data source inputs are available.
`minorSourceChangeSeed`	Contains an integer value that is incremented each time a data source is added or removed, or the data contained in any of the data sources changes. This provides a way for a codec to know if the data available to it has changed.
`sourceData`	Contains a pointer to a `CDSequenceDataSource` structure (page 233). This structure contains a linked list of all data sources. Because each data source contains a link to the next data source, a codec can access all data sources from this field.
`preferredPacketSizeInBytes`	Specifies the preferred packet size for data.
`requestedBufferWidth`	Specifies the the width of the image buffer to use, in pixels. For this value to be used, the `codecWantsSpecialScaling` flag in the `CodecCapabilities` record must be set.
`requestedBufferHeight`	Specifies the the height of the image buffer to use, in pixels. For this value to be used, the `codecWantsSpecialScaling` flag in the `CodecCapabilities` record must be set.

The Decompression Parameters Structure

Several fields have been added to the decompression parameters structure (`CodecDecompressParams`) originally documented in “The Decompression Parameters Structure” in *Inside Macintosh: QuickTime Components*, page 4-46.

In addition, there are several new flags in the `codecConditions` field. The first two fields listed below (`frameTime`, `reserved`) replace the last field (`reserved`) documented in Chapter 4 of *Inside Macintosh: QuickTime Components*.

```
struct CodecDecompressParams
{
...
```

```
/* The following fields are defined in QuickTime 2.0 or later */
    ICMFrameTimePtr     frameTime;      /* banddecompress */
    long                reserved[1];

    SInt8           matrixFlags;
    SInt8           matrixType;
    Rect            dstRect;    /*  only valid for simple transforms */

/* The following fields are defined in QuickTime 2.1 or later */
    UInt16          majorSourceChangeSeed;
    UInt16          minorSourceChangeSeed;
    CDSequenceDataSourcePtr sourceData;

    RgnHandle   maskRegion;

/* The following fields are defined in QuickTime 2.5 or later */
    OSType          **wantedDestinationPixelTypes;      /* Handle to
                                    0-terminated list of OSTypes */

    long            screenFloodMethod;
    long            screenFloodValue;
    short           preferredOffscreenPixelSize;

/* The following fields are defined in QuickTime 3 or later */
    boolean         enableBlackLining;
    long            requestedBufferWidth;
    long            requestedBufferHeight;
;
```

Field descriptions

`frameTime` Contains a pointer to an `ICMFrameTime` structure (page 232). This structure contains a frame’s time information for scheduled asynchronous decompression operations.

`matrixFlags` Flags specifying information about the transformation matrix. Currently, can be 0 or one of the following:

```
enum {
    matrixFlagScale2x = 1L<<7,
    matrixFlagScale1x = 1L<<6,
    matrixFlagScaleHalf = 1L<<5
};
```

`matrixType`	Contains the type of the transformation matrix, as returned by `GetMatrixType()`. (For additional information refer to *Inside Macintosh: QuickTime,* p. 2-342).
`dstRect`	The destination rectangle. The result of the source rectangle (`srcRect`) transformed by the transformation matrix (`matrix`).
`majorSourceChangeSeed`	Contains an integer value that is incremented each time a data source is added or removed. This provides a fast way for a codec to know when it needs to redetermine which data source inputs are available.
`minorSourceChangeSeed`	Contains an integer value that is incremented each time a data source is added or removed, or the data contained in any of the data sources changes. This provides a way for a codec to know if the data available to it has changed.
`sourceData`	Contains a pointer to a `CDSequenceDataSource` structure (page 233). This structure contains a linked list of all data sources. Because each data source contains a link to the next data source, a codec can access all data sources from this field.
`maskRegion`	If the `maskRegion` field is not `nil`, it contains a QuickDraw region that is equivalent to the bit map contained in the `maskBits` field. For some codecs, using the QuickDraw region may be more convenient than the mask bit map.
`wantedDestinationPixelTypes`	Filled in by the codec during `ImageCodecPreDecompress`. Contains a handle to a zero-terminated list of non-RGB pixels that the codec can decompress to. Leave set to `nil` if the codec does not support non-RGB pixel spaces. The ICM copies this data structure, so it is up to the codec to dispose of it later. Since the predecompress call can be called often, it is suggested that codecs allocate this handle during the Open routine and dispose of it during the Close routine.
`screenFloodMethod`	For codecs that require key-color flooding. One of:

```
enum {
    kScreenFloodMethodNone = 0,
    kScreenFloodMethodKeyColor = 1,
    kScreenFloodMethodAlpha = 2
};
```

`screenFloodValue`
: If `screenFloodMethod` is `kScreenFloodMethodKeyColor`, contains the index of the color that should be used to flood the image area on screen when an refresh occurs. This is valid for both indexed and direct screen devices (e.g., for 16 bit depth devices, it should contain the 5-5-5 RGB value). If `screenFloodMethod` is `kScreenFloodMethodAlpha`, contains the value that the alpha-channel should be flooded with.

`preferredOffscreenPixelSize`
: Should be filled in `ImageCodecPreDecompress` with the preferred depth of an offscreen buffer should the ICM have to create one. It is not guaranteed that an offscreen buffer will actually be of this depth. A codec should still be sure to specify what depths it can decompress to by using the capabilities field. A codec might use this field if if was capable of decompressing to several depths, but was faster decompressing to a particular depth.

`enableBlackLining`
: If true, indicates that the client has requested blacklining (displaying every other line of the image). Blacklining increases the speed of movie playback while decreasing the image quality.

`requestedBufferWidth`
: Specifies the the width of the image buffer to use, in pixels. For this value to be used, the `codecWantsSpecialScaling` flag in the `CodecCapabilities` record must be set.

`requestedBufferHeight`
: Specifies the the height of the image buffer to use, in pixels. For this value to be used, the `codecWantsSpecialScaling` flag in the `CodecCapabilities` record must be set.

codecConditions flags

Several new flags exist that can be set in the `codecConditions` parameter. They are:

```
enum {
    codecConditionFirstScreen       = 1L << 12,
    codecConditionDoCursor          = 1L << 13,
    codecConditionCatchUpDiff       = 1L << 14,
    codecConditionMaskMayBeChanged  = 1L << 15,
    codecConditionToBuffer          = 1L << 16
};
```

`codecConditionFirstScreen`

Indicates when the codec is decompressing an image to the first of multiple screens. That is, if the decompressed image crosses multiple screens, then the codec can look at this flag to determine if this is the first time an image is being decompressed for each of the screens to which it is being decompressed.

A codec that depends on the `maskBits` field of `decompressParams` being a valid `regionHandle` on `CDPreDecompress` needs to know that in this case it is not able to clip images since the region handle is only passed for the first of the screens; clipping would be incorrect for the subsequent screen for that image.

`codecConditionDoCursor`

Set to 1 if the decompressor component should shield and unshield the cursor for the current decompression operation. This flag is set only if the codec has indicated its ability to handle cursor shielding by setting the `codecCanShieldCursor` flag in the capabilities field during `CDPreDecompress`.

`codecConditionCatchUpDiff`

Indicates if the current frame is a “catch up” frame. Set this flag to 1 if the current frame is a catch-up frame. Note that you must also set the `codecFlagCatchUpDiff` flag to 1. This may be useful to decompressors that can drop frames when playback is falling behind.

`codecConditionMaskMayBeChanged`

The Image Compression Manager has always included support for decompressors that could provide a bit mask of pixels that were actually drawn when a particular frame was decompressed. If a decompressor can provide a bit mask of pixels that changed, the Image Compression Manager transfers to the screen only the pixels that actually changed.

QuickTime 2.1 extended this capability by adding a new condition flag (`codecConditionMaskMayBeChanged`) to the conditionFlags field of the decompression parameters structure. The decompressor should only write back the mask only when this flag is set. The flag is used only by ImageCodecBandDecompress (page 245).

codecConditionToBuffer
: Set to 1 if the current decompression operation is decompressing into an offscreen buffer.

Functions

ImageCodecExtractAndCombineFields

Performs field operations on video data. It allows fields from two separate images, compressed in the same format, to be combined in to a new compressed frame. Typically the operation results in an image of identical quality to the source images. Fields of a single image may also be duplicated or reversed by this function.

```
pascal ComponentResult ImageCodecExtractAndCombineFields (
                      ComponentInstance ci,
                      long fieldFlags,
                      void *data1,
                      long dataSize1,
                      ImageDescriptionHandle desc1,
                      void *data2,
                      long dataSize2,
                      ImageDescriptionHandle desc2,
                      void *outputData,
                      long *outDataSize,
                      ImageDescriptionHandle descOut);
```

ci	Specifies the image compressor component for the request.
fieldFlags	Flags specifying the operation to be performed. A correctly formed request will specify two input fields, mapping one to the odd output field and the other to the even output field. The following flags are defined:

`evenField1ToEvenFieldOut`
: Maps the even field specified by the `data1` parameter to the even output field.

`evenField1ToOddFieldOut`
: Maps the even field specified by the `data1` parameter to the odd output field.

`oddField1ToEvenFieldOut`
: Maps the odd field specified by the `data1` parameter to the even output field.

`oddField1ToOddFieldOut`
: Maps the odd field specified by the `data1` parameter to the odd output field.

`evenField2ToEvenFieldOut`
: Maps the even field specified by the `data2` parameter to the even output field.

`evenField2ToOddFieldOut`
: Maps the even field specified by the `data2` parameter to the odd output field.

`oddField2ToEvenFieldOut`
: Maps the odd field specified by the `data2` parameter to the even output field.

`oddField2ToOddFieldOut`
: Maps the odd field specified by the `data2` parameter to the odd output field.

`data1`	A pointer to a buffer containing the compressed image data for the first input field.
`dataSize1`	Specifies the size of the `data1` buffer.
`desc1`	An image description structure describing the format and characteristics of the data in the `data1` buffer.
`data2`	A pointer to a buffer containing the compressed image data for the second input field. Set to `nil` if the requested operation uses only one input frame.
`dataSize2`	Specifies the size of the `data2` buffer. Set to 0 if the requested operation uses only one input frame.

`desc2`	An image description structure describing the format and characteristics of the data in the `data2` buffer. Set to `nil` if the requested operation uses only one input frame.
`outputData`	A pointer to a buffer to receive the resulting frame.
`outDataSize`	On output this parameter returns the actual size of the new compressed image data.
`descOut`	Specifies the desired format of the resulting frames. Typically this is the same format specified by the `desc1` and `desc2` parameters.

DISCUSSION

This codec routine implements the functionality of the `ImageFieldSequenceExtractCombine` function described in Chapter 3, "Image Compression Manager." If your codec is capable of separately compressing both fields of a video frame, you should incorporate support for this function.

Your codec must ensure that it understands the image formats specified by `desc1` and `desc2` parameters, as these may not be the same as the codec's native image format. The image format specified by the `descOut` parameter will be the same as the codec's native image format.

The component selector for this function is:

```
kImageCodecExtractAndCombineFieldsSelect    = 0x0015
```

ImageCodecPreDecompress

Your component receives an `ImageCodecPreDecompress` call before decompressing an image or sequence of frames.

```
pascal ComponentResult ImageCodecPreDecompress(
                          ComponentInstance ci,
                          CodecDecompressParams *params);
```

`ci`	Specifies the image decompressor component for the request.

`params` Contains a pointer to a decompression parameters structure. See "The Decompression Parameters Structure" (*Inside Macintosh: QuickTime Components*, page 4-46), and "The Decompression Parameters Structure" (page 237) in this chapter for a complete description.

DISCUSSION

If your decompressor component supports scheduled asynchronous decompression operations, be sure to set the `codecCanAsyncWhen` flag to 1 in the `flags` field of your component's compressor capabilities structure. If you set `codecCanAsyncWhen` you must also set `codecCanAsync`. Codecs that support scheduled asynchronous decompression are strongly advised to also set the `codecCanShieldCursor` flag.

If your decompressor component uses a secondary hardware buffer for its images, be sure to set the `codecHasVolatileBuffer` flag to 1 in the `flags` field of your component's compressor capabilities structure.

If your decompressor component is used solely as a transfer codec and uses the `CDCodecNewImageBufferMemory` call to create an offscreen buffer that is really onscreen, your codec will need to set the `codecImageBufferIsOnScreen` flag to 1.

See the section "The Compressor Capability Structure" (page 234) for more information about these flags.

ImageCodecBandDecompress

Your component receives an `ImageCodecBandDecompress` call to decompress a frame.

```
pascal ComponentResult ImageCodecBandDecompress(
                        ComponentInstance ci,
                        CodecDecompressParams *params);
```

`ci` Specifies the image decompressor component for the request.

`params`	Contains a pointer to a decompression parameters structure. See "The Decompression Parameters Structure" (*Inside Macintosh: QuickTime Components*, page 4-46), and "The Decompression Parameters Structure" (page 237) in this chapter for a complete description.

DISCUSSION

For scheduled asynchronous decompression operations, the Image Compression Manager supplies a reference to an `ICMFrameTime` structure in this function's decompression parameters structure parameter. The `ICMFrameTime` structure (page 232) contains time information governing the scheduled decompression operation, including the time at which the frame must be displayed. For synchronous or immediate asynchronous decompress operations, the frame time is set to `nil`.

When your component has finished the decompression operation, it must call the completion function. In the past, for asynchronous operations, your component called that function directly. For scheduled asynchronous decompression operations, your component should call the Image Compression Manager's `ICMDecompressComplete` function(page 258).

If your component set the `codecCanAsyncWhen` flag in pre-decompress but cannot support scheduled asynchronous decompression for a given frame, it must return an error code of `codecCantWhenErr`. If your component's queue is full, it should return an error code of `codecCantQueueErr`.

ImageCodecFlush

Empties a image decompressor component's input queue of pending scheduled frames.

```
pascal ComponentResult ImageCodecFlush(
                    ComponentInstance ci);
```

`ci`	Specifies the image decompressor component for the request.

DISCUSSION

Your component receives the `ImageCodecFlush` function whenever the Image Compression Manager needs to cancel the display of all scheduled frames.

Your decompressor should empty its queue of scheduled asynchronous decompression requests. For each request, your component must call the `ICMDecompressComplete` function. Be sure to set the `err` parameter to –1, indicating that the request was canceled. Also, you must set both the `codecCompletionSource` and `codecCompletionDest` flags to 1. Only decompressor components that support scheduled asynchronous decompression receive this call.

SPECIAL CONSIDERATIONS

Your component's `ImageCodecFlush` function may be called at interrupt time.

ImageCodecSetTimeCode

Sets the timecode for the next frame that is to be decompressed.

```
pascal OSErr ImageCodecSetTimeCode (
                     ComponentInstance ci,
                     void *timeCodeFormat,
                     void *timeCodeTime);
```

`ci`	Specifies the image decompressor component for the request.
`timeCodeFormat`	Contains a pointer to a timecode definition structure. This structure contains the timecode definition information for the next frame to be decompressed.
`timeCodeTime`	Contains a pointer to a timecode record structure. This structure contains the time value for the next frame in the current sequence.

DISCUSSION

Your component receives `CDCodecSetTimeCode` function whenever an application calls the Image Compression Manager's `SetDSequenceTimeCode` function. That function allows an application to set the timecode for a frame that is to be decompressed.

The timecode information you receive applies to the next frame to be decompressed and is provided to the decompressor in the `CDBandDecompress` function.

ImageCodecIsImageDescriptionEquivalent

Compares image descriptions.

```
pascal ComponentResult ImageCodecIsImageDescriptionEquivalent (
                        ComponentInstance ci,
                        ImageDescriptionHandle newDesc,
                        Boolean *equivalent);
```

`ci`	Specifies the image compressor component for the request.
`newDesc`	Contains a handle to the image description structure that describes the compressed image.
`equivalent`	Contains a pointer to a Boolean value. If the `ImageDescriptionHandle` provided in the `newDesc` parameter is equivalent to the image description currently in use by the image sequence, this value is set to `true`. If the `ImageDescriptionHandle` is not equivalent, and therefore a new image sequence must be created to display an image using the new image description, this value is set to `false`.

DISCUSSION

Your component receives the `ImageCodecIsImageDescriptionEquivalent` request whenever an application calls the Image Compression Manager's `CDSequenceEquivalentImageDescription` function (page 201). Implementing this function can significantly improve playback of edited video sequences using your codec. For example, if two sequences are compressed at different quality levels and are edited together they will have different image descriptions because their quality values will be different. This will force QuickTime to use two separate decompressor instances to display the images. By implementing this function your decompressor can tell QuickTime that differences in quality levels don't require separate decompressors. This saves memory and time, thus improving performance.

SPECIAL CONSIDERATIONS

The current image description is not passed in this function because the Image Compression Manager assumes the codec has already made copies of all relevant data fields from the current image description during the `ImageCodecPreDecompress` call.

ImageCodecNewMemory

Requests codec-allocated memory. Some hardware codecs may have on-board memory that can be used to store compressed and/or decompressed data. `ImageCodecNewMemory` makes this memory available for use by clients of the codec. Some software codecs may be able to optimize thier performance by having more control over memory allocation. `ImageCodecNewMemory` makes this control available.

```
pascal ComponentResult ImageCodecNewMemory (
                        ComponentInstance ci,
                        Ptr *data,
                        Size dataSize,
                        long dataUse,
                        ICMMemoryDisposedUPP memoryGoneProc,
                        void *refCon);
```

`ci`	Specifies the image decompressor component for the request.
`data`	Returns a pointer to the allocated memory.
`dataSize`	Specifies the desired size of the `data` buffer.
`dataUse`	A code that indicates how the memory is to be used. For example, the memory may be used to store compressed data before it's displayed, mask plane data, or decompressed data.

If there is no benefit to storing a particular kind of data in codec memory, the codec should refuse the request for the memory allocation. The defined values are for data use are:

0x00000001	Memory will be used for holding compressed image data.
0x00000002	Memory will be used for an offscreen image buffer.

`memoryGoneProc`

A pointer to a function that will be called before disposing of the memory allocated by a codec. Your callback function must be in the following form:

```
pascal void (*ICMMemoryDisposedProcPtr)
              (Ptr memoryBlock, void *refcon);
```

This function must be called if the memory block is to be disposed of by the codec instead of by `ImageCodeDisposeMemory`. For example, this would occur if the codec is closed and still has memory allocation outstanding or if the memory is required to complete another operation. The `memoryGoneProc` must not be called at interrupt time.

`refCon` Contains a reference constant value that your codec must pass to the `memoryGoneProc` function.

DISCUSSION

Your component receives the `ImageCodecNewMemory` request whenever an application calls the Image Compression Manager's `CDSequenceNewMemory` function (page 202).

SPECIAL CONSIDERATIONS

The Image Compression Manager does not currently track memory allocations. When a compressor or decompressor component instance is closed, it must ensure that all blocks allocated by that instance are disposed (and call the `ICMMemoryDisposeUPP`). If your codec does not currently have free memory for compression frame data, but will soon , you can return `codecMemoryFullPleaseWait` to indicate this fact.

ImageCodecNewImageBufferMemory

Requests the codec to allocate memory for an offscreen buffer of non-RGB pixels. This call is used to support a codec decompressing into a non-RGB buffer. The transfer codec is responsible for defining the offscreen and transferring the image from the offscreen to the destination.

```
pascal ComponentResult ImageCodecNewImageBufferMemory(
                    ComponentInstance ci,
                    CodecDecompressParams *params,
                    long flags,
                    ICMMemoryDisposedUPP memoryGoneProc,
                    void *refCon);
```

`ci` Specifies the image decompressor component for the request.

`params` Contains a pointer to a decompression parameters structure. See “The Decompression Parameters Structure” (*Inside Macintosh: QuickTime Components*, page 4-46), and “The Decompression Parameters Structure” (page 237) in this chapter for a complete description. Your codec must fill in the `dstPixMap.baseAddr` and the `dstPixMap.rowBytes` fields in this structure.

`flags` Currently, this parameter is always set to 0.

`memoryGoneProc`

A pointer to a function that will be called before disposing of the memory allocated by a codec. Your callback function must be in the following form:

```
pascal void (*ICMMemoryDisposedProcPtr)
             (Ptr memoryBlock, void *refcon);
```

This function must be called if the memory block is to be disposed of by the codec instead of by `ImageCodeDisposeMemory`. For example, this would occur if the codec is closed and still has memory allocation outstanding or if the memory is required to complete another operation. The `memoryGoneProc` must not be called at interrupt time.

`refCon` Contains a reference constant value that your codec must pass to the `memoryGoneProc` function.

DISCUSSION

Your component receives the `ImageCodecNewImageBufferMemory` request whenever another codec has requested a non-RGB offscreen buffer of the type of your component's subtype. See "Accelerated Video Support" (page 225) for more information.

SPECIAL CONSIDERATIONS

The Image Compression Manager does not currently track memory allocations. When a compressor or decompressor component instance is closed, it must ensure that all blocks allocated by that instance are disposed (and call the `ICMMemoryDisposeUPP`).

ImageCodecDisposeMemory

Disposes codec-allocated memory.

```
pascal ComponentResult ImageCodecDisposeMemory (
                     ComponentInstance ci,
                     Ptr data);
```

`ci`	Specifies the image compressor component for the request
`data`	Points to the previously allocated memory block.

DISCUSSION

Your component receives the `ImageCodecDisposeMemory` request whenever an application calls the Image Compression Manager's `CDSequenceDisposeMemory` function (page 204).

SPECIAL CONSIDERATIONS

When a codec instance is closed, it must ensure that all blocks allocated by that instance are disposed (and call the `ICMMemoryDisposeUPP`).

ImageCodecRequestSettings

Displays a dialog box containing codec-specific compression settings.

```
pascal ComponentResult ImageCodecRequestSettings (
                        ComponentInstance ci,
                        Handle settings,
                        Rect *rp,
                        ModalFilterUPP filterProc);
```

`ci`	Specifies the image compressor component for the request.
`settings`	A handle of data specific to the codec. If the handle is empty, the codec should use its default settings.
`rp`	A pointer to a rectangle giving the coordinates of the standard compression dialog box in global screen coordinates. The codec can use this to position its dialog box in the same area of the screen.
`filterProc`	A pointer to a modal dialog filter procedure that the codec must either pass to the `ModalDialog` function or call at the beginning of the codec dialog filter. This procedure gives the calling application and standard compression dialog box a chance to process update events.

DISCUSSION

The `ImageCodecRequestSettings` function allows the display of a dialog box of additional compression settings specific to the codec. These settings are stored in a `settings` handle. The codec can store any data in any format it wants in the `settings` handle and resize it accordingly. It should store some type of tag or version information that it can use to verify that the data belongs to the codec. The codec should not dispose of the handle.

ImageCodecGetSettings

Returns the settings chosen by the user.

```
pascal ComponentResult ImageCodecGetSettings (
                        ComponentInstance ci,
                        Handle settings);
```

`ci`	Specifies the image compressor component for the request.
`settings`	A handle that the codec should resize and fill in with the current internal settings. If there are no current internal settings, resize it to 0. Don't dispose of this handle.

DISCUSSION

The `ImageCodecGetSettings` function returns the codec's current internal settings. If there are no current settings or the settings are the same as the defaults, the codec can set the handle to `nil`.

ImageCodecSetSettings

Sets the settings of the optional dialog box.

```
pascal ComponentResult ImageCodecSetSettings (
                        ComponentInstance ci,
                        Handle settings);
```

`ci`	Specifies the image compressor component for the request. Applications obtain this reference from the Component Manager's `OpenComponent` function.
`settings`	A handle to internal settings originally returned by either `ImageCodecRequestSettings` or `ImageCodecGetSettings`. The codec should set its internal settings to match those of the settings handle. Because the codec does not own the handle, it should not dispose of it and should copy only its contents, not the handle itself. If the settings handle passed is empty, the codec should sets its internal settings to a default state.

DISCUSSION

The `ImageCodecSetSettings` function allows a codec to return its private settings. Set the codec's internal settings to the state specified in the settings handle. The codec should always check the validity of the contents of the handle so that invalid settings are not used.

ImageCodecHitTestData

This routine is called when the application calls `PtInDSequenceData`. It returns a Boolean indicating whether or not the specified point is contained within the specified image data.

```
pascal ComponentResult ImageCodecHitTestData(
                        ComponentInstance ci,
                        ImageDescriptionHandle desc,
                        void *data,
                        Size dataSize,
                        Point where,
                        Boolean *hit);
```

`ci`	Specifies the image decompressor component for the request.
`desc`	Contains an `ImageDescriptionHandle` for the image data pointed to by the `data` param.
`data`	Pointer to compressed data in the format specified by the `desc` param.
`dataSize`	Size of the compressed data referred to by the `data` param.
`where`	A QuickDraw `Point` (0,0) based at the top-left corner of the image.
`hit`	A pointer to a Boolean. The Boolean should be set to `true` if the point specified by the `where` parameter is contained within the compressed image data specified by the `data` param.

DISCUSSION

The `ImageCodecHitTestData` function allows the calling application to perform hit testing on compressed data. The codec should set the hit parameter to true if the compressed data contains data at the point specified by the where parameter. The hit parameter should be set to false if the specified point falls within a blank portion of the image.

ImageCodecGetMaxCompressionSizeWithSources

Your codec receives the request when an application calls the Image Compression Manager's `GetCSequenceMaxCompressionSize` function. The caller uses this function to determine the maximum size the data will be compressed to for a given image and set of data sources.

```
pascal ComponentResult ImageCodecGetMaxCompressionSizeWithSources(
                        ComponentInstance ci,
                        PixMapHandle src,
                        const Rect *srcRect,
                        short depth,
                        CodecQ quality,
                        CDSequenceDataSourcePtr sourceData,
                        long *size);
```

`ci`	Specifies the image decompressor component for the request.
`src`	Contains a handle to the source image. The source image is stored in a pixel map structure. Applications use the size information you return to allocate buffers for more than one image. Consequently, your compressor should not consider the contents of the image when determining the maximum compressed size. Rather, you should consider only the quality level, pixel depth, and image size.
	This parameter may be set to `nil`. In this case the application has not supplied a source image—your component should use the other parameters to determine the characteristics of the image to be compressed.
`srcRect`	Contains a pointer to a rectangle defining the portion of the source image to compress.

`depth`	Specifies the depth at which the image is to be compressed. Values of 1, 2, 4, 8, 16, 24, and 32 indicate the number of bits per pixel for color images. Values of 33, 34, 36, and 40 indicate 1-bit, 2-bit, 4-bit, and 8-bit grayscale, respectively, for grayscale images.
`quality`	Specifies the desired compression image quality. See the chapter "Image Compression Manager" in *Inside Macintosh: QuickTime* for valid values.
`sourceData`	Contains a pointer to a `CDSequenceDataSource` structure (page 233). This structure contains a linked list of all data sources. Because each data source contains a link to the next data source, a codec can access all data sources from this field.
`size`	Contains a pointer to a field to receive the maximum size, in bytes, of the compressed image.

DISCUSSION

The `ImageCodecGetMaxCompressionSizeWithSources` function is similar in purpose to the `ImageCodecGetMaxCompressionSize` function documented in *Inside Macintosh: QuickTimeComponents* (page 4-55). This function, however, also considers data sources that the codec may compress with the image.

ImageCodecSourceChanged

Your codec receives this notification that one of the data sources has changed when an application calls the Image Compression Manager's `CDSequenceSetSourceData` or `CDSequenceChangedSourceData` functions.

```
pascal ComponentResult ImageCodecSourceChanged(
                    ComponentInstance ci,
                    UInt32 majorSourceChangeSeed,
                    UInt32 minorSourceChangeSeed,
                    CDSequenceDataSourcePtr sourceData,
                    long *flagsOut);
```

`ci`	Specifies the image decompressor component for the request.

`majorSourceChangeSeed`
: Contains an integer value that is incremented each time a data source is added or removed. This provides an easy way for a codec to know when it needs to redetermine which data source inputs are available.

`minorSourceChangeSeed`
: Contains an integer value that is incremented each time a data source is added or removed, or the data contained in any of the data sources changes. This provides a way for a codec to know if the data available to it has changed.

`sourceData`
: Contains a pointer to a `CDSequenceDataSource` structure (page 233). This structure contains a linked list of all data sources. Because each data source contains a link to the next data source, a codec can access all data sources from this field.

`flagsOut`
: Contains a pointer to a `CDSequenceDataSource` structure (page 233). This structure contains a linked list of all data sources. Because each data source contains a link to the next data source, a codec can access all data sources from this field.

DISCUSSION

This routine is provided to notify the codec component that one of the data sources has changed.

Image Compression Manager Utility Functions

ICMShieldSequenceCursor

Hides the cursor during decompression operations.

```
pascal OSErr ICMShieldSequenceCursor (ImageSequence seqID);
```

`seqID`
: Identifies the sequence for which to shield the cursor.

DISCUSSION

Your component may call the `ICMShieldSequenceCursor` function to manage the display of the cursor during decompression operations.

For correct image display behavior, the cursor must be shielded (hidden) during decompression. By default, the Image Compression Manager handles the cursor for you, hiding it at the beginning of a decompression operation and revealing it at the end.

With the advent of scheduled asynchronous decompression, however, the ICM cannot do as precise a job of managing the cursor, because it does not know exactly when scheduled operations actually begin and end. While the ICM can still manage the cursor, it must hide the cursor when each request is queued, rather than when the request is serviced. This may result in the cursor remaining hidden for long periods of time.

In order to achieve better cursor behavior, you can choose to manage the cursor in your decompressor component. If you so choose, you can use the `ICMShieldSequenceCursor` function to hide the cursor—the ICM displays the cursor when you call the `ICMDecompressComplete` function. In this manner, the cursor is hidden only when your component is decompressing and displaying the frame.

SPECIAL CONSIDERATIONS

This function may be called at interrupt time.

ICMDecompressComplete

Signals the completion of a decompression operation.

```
pascal void ICMDecompressComplete (
                    ImageSequence seqID,
                    OSErr err,
                    short flag,
                    ICMCompletionProcRecordPtr completionRtn);
```

`seqID`	Identifies the frame’s sequence.

`err` Indicates whether the operation succeeded or failed. Set this parameter to 0 for successful operations. For failed operations, set the error code appropriate for the failure. For canceled operations (for example, when the ICM calls your component's `ImageCodecFlush` function), set this parameter to –1.

`flag` Completion flags. Note that you may set more than one of these flags to 1. The following flags are defined:

`codecCompletionSource`
Your component is done with the source buffer. Set this flag to 1 when you are done with the processing associated with the source buffer.

`codecCompletionDest`
Your component is done with the destination buffer. Set this flag to 1 when you are done with the processing associated with the destination buffer.

`codecCompletionDontUnshield`
Set this flag to 1 when you do not want the ICM to unshield the cursor as it normally would. Only codecs that are completely managing the cursor themselves should set this flag. See the section "Hardware Cursors" (page 224) for more information.

`completionRtn`
Contains a pointer to a completion function structure. That structure identifies the application's completion function, and contains a reference constant associated with the frame. Your component obtains the completion function structure as part of the decompression parameters structure provided by the Image Compression Manager at the start of the decompression operation.

DISCUSSION

Your component must call this function at the end of decompression operations.

SPECIAL CONSIDERATIONS

Prior to QuickTime 2.0, decompressor components called the application's completion function directly. For compatibility, that method is still supported except for scheduled asynchronous decompression operations, which must use the `ICMDecompressComplete` call. Newer decompressors should always use `ICMDecompressComplete` rather than calling the completion function directly, regardless of the type of decompression operation.

Image Transcoder Components

This chapter is primarily for application developers who wish to write image transcoders.

To use transcoders, refer to Chapter 3, “Image Compression Manager.”

Prior to QuickTime 2.5, if you needed to convert compressed image data into another compressed image format, it was necessary to decompress the compressed image data to RGB pixels, and then compress the RGB pixels into the new format. For some types of compressed image data, it is now possible to convert directly from one compressed format to another. This direct conversion process is called **image transcoding**.

About Image Transcoding

Transcoding has two distinct advantages over the decompress-then-recompress approach to converting the format of compressed data. The first advantage is that the operation is usually substantially faster, since much of the data can be copied directly from the source image data format to the destination image data format. The second advantage is that the operation is usually more accurate because decompressing and recompressing provides two steps for introducing rounding and quantization errors. By directly transcoding, opportunities for small errors are substantially reduced.

Image Transcoding Support

QuickTime’s image transcoding support is contained within the Image Compression Manager. Image transcoding can be invoked either explicitly, using new APIs in the Image Compression Manager, or implicitly, by using existing routines for decompressing images.

QuickTime's support for decompressing images has been enhanced so that if a request is issued to decompress an image, but no image decompressor component is installed for that image format, QuickTime will attempt to locate an image transcoder to convert the image data into a supported format. This transcoding is performed transparently to the calling application. This automatic image transcoding is supported for both QuickTime movies and compressed image data stored in QuickDraw pictures.

QuickTime also provides an API for applications to transcode images. These APIs make it possible for any application to take compressed image data and transcode it into another format. This capability is useful for applications that create QuickTime movies by combining segments of other QuickTime movies. These applications often convert the format of the compressed image data by decompressing the image and then recompressing it to the new format. If no other processing is to be performed on the compressed data, you can use an image transcoder to increase the speed and fidelity of the operation.

As with most other services in QuickTime, the details of image transcoding are handled by components. The Image Compression Manager uses image transcoder components to perform both implicit and explicit image transcoding. Application developers that perform image transcoding interact with the Image Compression Manager, not directly with the image transcoder components themselves. The Image Compression Manager takes care of the details of working with image transcoder components. If you want to add new image transcoding operations to QuickTime, you can write an image transcoder component.

Using Image Transcoder Components

The Image Compression Manager uses an image sequence when compressing or decompressing data. An image sequence allows QuickTime to make certain optimizations because it knows that a similar operation will be repeated multiple times (that is, images will be repeatedly compressed to the same image data format). Similarly, the Image Compression Manager's support for image transcoding is based on an image transcoding sequence. The image transcode sequence identifier is an opaque value as shown below.

```
typedef long ImageTranscodeSequence;
```

All transcoding functions are described in Chapter 3, “Image Compression Manager.” Image transcoder component functions are described later in this chapter.

To create an image transcoding sequence, use the `ImageTranscodeSequenceBegin` function. To transcode a frame of image data, use the `ImageTranscodeFrame` function. The caller of this routine is responsible for disposing of the transcoded data returned by `ImageTranscodeFrame` using the `ImageTranscodeDisposeFrameData` routine.

When the transcoded image data returned by `ImageTranscodeFrame` is no longer needed, call `ImageTranscodeDisposeFrameData` to dispose of the data. When an image transcoding sequence is complete, use `ImageTranscodeSequenceEnd` to dispose of the image transcoding sequence.

Creating an Image Transcoder Component

It is only necessary to understand image transcoder components if you are writing an image transcoder. To perform image transcoding, you should use the services provided by the Image Compression Manager.

Image transcoder components are standard Component Manager components. For details on creating components, see *Mac OS For QuickTime Programmers.*

Image transcoder components have a type of ‘`imtc`’ as defined below.

```
enum {
    ImageTranscodererComponentType = 'imtc'
};
```

The subtype field of the component defines the compressed image data format that the transcoder accepts as an input. The manufacturer field of the component defines the compressed image data format that the transcoder generates as output. For example, a trancoder from Motion JPEG Format A to Motion JPEG Format B would have a subtype of `'mjpg'` and a manufacturer code of `'mjpb'`. No component-specific flags are currently defined for transcoders; they should be set to 0. Each transcoder component function is described in “Image Transcoder Components Reference” in this chapter.

Example Image Transcoder Component

The example code in Listing 5-1 shows an image transcoder component. It converts an imaginary compressed data format 'bgr' to uncompressed RGB pixels. The transcoding process simply copies the source data to the destination and inverts each byte in the process. This example shows the format of how an image transcoder might work without getting into the details of a particular image transcoding operation.

Listing 5-1 An image transcoder component, converting a compressed data format to uncompressed RGB pixels

```
#include <ImageCompression.h>
pascal ComponentResult main(ComponentParameters *params, Handle storage
);
pascal ComponentResult TestTranscoderBeginSequence (Handle storage,
ImageDescriptionHandle srcDesc, ImageDescriptionHandle *dstDesc, void
*data, long dataSize);

pascal ComponentResult TestTranscoderConvert (Handle storage, void
*srcData, long srcDataSize, void **dstData, long *dstDataSize);

pascal ComponentResult TestTranscoderDisposeData (Handle storage, void
*dstData);

pascal ComponentResult TestTranscoderEndSequence (Handle storage);

pascal ComponentResult main(ComponentParameters *params, Handle storage )
{
    ComponentFunctionUPP proc = nil;
    ComponentResult err = noErr;

    switch (params->what) {
        case kComponentOpenSelect:
        case kComponentCloseSelect:
            break;
        case kImageTranscoderBeginSequenceSelect:
            proc = (ComponentFunctionUPP) TestTranscoderBeginSequence;
            break;
        case kImageTranscoderConvertSelect:
            proc = (ComponentFunctionUPP)TestTranscoderConvert;
            break;
        case kImageTranscoderDisposeDataSelect:
            proc = (ComponentFunctionUPP) TestTranscoderDisposeData;
            break;
        case kImageTranscoderEndSequenceSelect:
            proc = (ComponentFunctionUPP) TestTranscoderEndSequence;
            break;
        default:
            err = badComponentSelect;
            break;
    }

    if (proc)
        err = CallComponentFunctionWithStorage(storage,
                        params, proc);

    return err;
}

pascal ComponentResult TestTranscoderBeginSequence (Handle storage,
ImageDescriptionHandle srcDesc, ImageDescriptionHandle *dstDesc, void
*data, long dataSize)
{
    *dstDesc = srcDesc;
    HandToHand((Handle *)dstDesc);
    (***dstDesc).cType = 'raw ';

    return noErr;
}

pascal ComponentResult TestTranscoderConvert (Handle storage, void
*srcData, long srcDataSize, void **dstData, long *dstDataSize)
{
    Ptr p;
    OSErr err;

    if (!srcDataSize)
        return paramErr;

    p = NewPtr(srcDataSize);
    err = MemError();
    if (err) return err;
    {
    Ptr p1 = srcData, p2 = p;
    long counter = srcDataSize;
    while (counter--)
        *p2++ = ~*p1++;
    }

    *dstData = p;
    *dstDataSize = srcDataSize;

    return noErr;
}

pascal ComponentResult TestTranscoderDisposeData (Handle storage, void
*dstData)
{
    DisposePtr((Ptr)dstData);

    return noErr;
}

pascal ComponentResult TestTranscoderEndSequence (Handle storage)
{
    return noErr;
}
```

Image Transcoder Components Reference

Functions

QuickTime 2.5 added four image transcoder component functions.

ImageTranscoderBeginSequence

Initiates an image transcoding sequence and specifies the input data format.

```
pascal ComponentResult ImageTranscoderBeginSequence (
                    ImageTranscoderComponent itc,
                    ImageDescriptionHandle srcDesc,
                    ImageDescriptionHandle *dstDesc,
                    void *data,
                    long dataSize);
```

`itc`	The image transcoder component.
`srcDesc`	The image description for the source compressed image data.
`dstDesc`	Returns a new image description.
`data`	First frame of data to be transcoded (may be nil).
`dataSize`	Size of compressed image data pointed to by the data.

DISCUSSION

The `ImageTranscoderBeginSequence` function specifies the format of source compressed image data in the `srcDesc` parameter. The image transcoder should allocate a new image description and return it in the `dstDesc` parameter. The new image description should be a completely filled out image description which is sufficient for correctly decompressing the data generated by subsequent calls to `ImageTranscoderConvert`.

ImageTranscoderConvert

Performs image transcoding operations.

```
pascal ComponentResult ImageTranscoderConvert (
                    ImageTranscoderComponent itc,
                    void *srcData,
                    long srcDataSize,
                    void **dstData,
                    long *dstDataSize);
```

`itc`	The image transcoder component.
`srcData`	Contains a pointer to the source compressed image data to transcode.
`srcDataSize`	Indicates the size of the source image data, in bytes.
`dstData`	Returns a pointer to the transcoded data.
`dstDataSize`	Returns the size of the transcoded data, in bytes.

DISCUSSION

The image transcoder component is responsible for allocating storage for the transcoded data, transcoding the data, and returning a pointer to the transcoded data in the `dstData` parameter. The size of the transcoded data in bytes should be returned in the `dstDataSize` parameter. The caller is responsible for disposing of the transcoded data using the `ImageTranscoderDisposeData` function.

The memory allocated to store the transcoded image data must not be in an unlocked handle. Even if the image transcoding operation can be performed in place, the transcoded data must be placed in a separate block of memory from the source data. The image transcoder component must not write back into the source image data.

The responsibility for allocating the buffer for the transcoded data has been placed in the transcoder with the intent that some hardware manufacturers may find it useful to place the transcoded data directly into on-board memory on their video board. If the transcoding operation is being performed on a QuickTime movie, the transcoded data pointer will be almost immediately passed on to a decompressor. If the decompressor is implemented in hardware, performance may be increased because the transcoded data is already loaded onto the decompression hardware.

ImageTranscoderDisposeData

Disposes of transcoded data.

```
pascal ComponentResult ImageTranscoderDisposeData (
                     ImageTranscoderComponent itc,
                     void *dstData);
```

`itc`	The image transcoder component.
`dstData`	Contains a pointer to the transcoded data.

DISCUSSION

When the client of the image transcoder component is done with a piece of transcoded data, `ImageTranscoderDisposeData` must be called with a pointer to the transcoded data. The image transcoder component should not make any assumptions about the maximum number of outstanding pieces of transcoded data or the order in which the transcoding data will be disposed.

ImageTranscoderEndSequence

Ends an image transcoding sequence.

```
pascal ComponentResult ImageTranscoderEndSequence
                     (ImageTranscoderComponent itc);
```

`itc`	The image transcoder component whose transcoder sequence is ending.

DISCUSSION

`ImageTranscoderEndSequence` is called when there are no more frames of data to be transcoded using the parameters specified in the previous call to `ImageTranscoderBeginSequence`. After calling this function the component will either be closed or receive another call to `ImageTranscoderBeginSequence` with a different image description. (For example, the dimensions of the source image may be different.)

CHAPTER 6

Movie Controller Components

This chapter discusses new features and changes to movie controller components as documented in Chapter 2 of *Inside Macintosh: QuickTime Components*.

New Features of Movie Controller Components

The new movie controller component features include five new actions and one new function. Additionally, one new flag has been defined to be returned by `MCGetControllerInfo` function.

Using Movie Controller Components

Changing the Shape of the Cursor

Many applications change the shape of the cursor depending on what it's currently over. The standard movie controller never changes the cursor, but other movie controllers may need to. An example of this is the QuickTime VR movie controller. Unfortunately, many applications need to control the cursor themselves—when a movie controller changes the cursor, these applications change it back immediately.

A simple solution is for applications to change the cursor only when it's first placed over a movie. (To determine whether a pointer is over the movie, use `mcPointInMovieController`.) After that, let the movie controller control the cursor until it exits the area over the movie. To give the movie controller the opportunity to change the cursor's shape, you must call either `MCIsPlayerEvent` or `MCIdle` while the cursor is over the movie, even if the movie is stopped. You can use the `mcActionSetCursorSettingEnabled` action to disable changes by movie controllers.

Adding a Custom Button

Your application can request a custom button in a movie controller by setting the `mcFlagsUseCustomButton` flag in the movie controller flags. The appearance of this button is determined by the movie controller. In the movie controller provided with QuickTime, this button is a downward-pointing arrow, and it is recommended that other movie controllers use a downward-pointing arrow for their custom buttons.

When its custom button is clicked, the movie controller generates the `mcActionCustomButtonClick` action. Your application responds by checking for the `mcActionCustomButtonClick` action in its action filter function and performing any tasks that are necessary.

Movie Controller Components Reference

This section describes the new constants and functions associated with movie controller components.

The movie controller has always supported actions for setting the selection. QuickTime 2.5 added two new actions for getting the selection. In addition, QuickTime 2.1 added one new action that prerolls the movie before playing it, and two more actions that enable your application to control whether the movie controller can change the cursor.

Movie Controller Actions

This section discusses new actions, which are integer constants (defined by the `mcAction` data type) used by movie controller components. Applications that use movie controller components can invoke these actions by calling the `MCDoAction` function.

This section does not describe all the existing constants documented in *Inside Macintosh: QuickTime Components*. The new constants are these:

```
enum {
    mcActionGetSelectionBegin               = 53,   /* param is TimeRecord */
    mcActionGetSelectionDuration            = 54,   /* param is TimeRecord */
    mcActionPrerollAndPlay                  = 55,   /* param is Fixed, play rate */
    mcActionGetCursorSettingEnabled         = 56,   /* param is pointer to Boolean */
    mcActionSetCursorSettingEnabled         = 57,   /* param is Boolean */
    mcActionCustomButtonClick               = 60,   /* param is pointer to */
                                                    /*   EventRecord */
    mcActionSetColorTable                   = 58,   /* param is CTabHandle */
};
typedef short mcAction;
```

Actions for Use by Applications

`mcActionGetSelectionBegin`

The parameter must contain a pointer to a time structure. The time returned is in the time scale of the movie. The returned time indicates the start time of the current user time selection.

`mcActionGetSelectionDuration`

The parameter must contain a pointer to a time structure. The time value returned is in the time scale of the movie. The returned time indicates the duration of the current user time selection. If there is no selection, this value is 0.

`mcActionPrerollAndPlay`

Your application can use this action to preroll a movie and then immediately play it. You should use this action whenever a movie controller is used and the movie needs to be played programmatically.

The parameter data must contain a fixed value that indicates the rate of play. Values greater than 0 correspond to forward rates; values less than 0 play the movie backward. A value of 0 stops the movie.

`mcActionGetCursorSettingEnabled`

Your application can use this action to determine whether cursor switching is enabled for a movie controller.

The parameter data must contain a pointer to a Boolean value—a value of `true` indicates that cursor switching is enabled. By default, this value is set to `true`.

`mcActionSetCursorSettingEnabled`

Your application can use this action to control whether the movie controller can change the cursor.

The parameter data must contain a Boolean value. Set this value to true to enable the movie controller to change the cursor. Set it to false to disable cursor switching.

Some movie controllers (QuickTime VR, for example) change the cursor while the pointer is over the movie to indicate that the pointer is over a hot spot. If you do not want the movie controller to change the cursor, you should use this action to prevent the movie controller from changing the cursor.

`mcActionSetColorTable`

Your application can use this action to determine when the movie controller is going to set a new color table. Setting a color table causes the window's palette to be updated to the new color table. Applications can use this action to monitor or control the movie controller's current color environment.

`mcActionSetFlags`

Your application can use this action to set a movie's control flags. The parameter data must contain a long integer that contains the new control flag values. An additional flag has been defined:

`mcFlagsUseCustomButton`

Requests a custom button in the movie controller. The appearance of this button is determined by the movie controller. In the movie controller provided with QuickTime, this button is a downward-pointing arrow, and it is recommended that other movie controllers use a downward-pointing arrow for their custom buttons. When its custom button is clicked, the movie controller generates the `mcActionCustomButtonClick` action. Your application responds by checking for the `mcActionCustomButtonClick` action in its action filter function and performing any tasks that are necessary.

Actions for Use by Action-Filter Functions

`mcActionCustomButtonClick`
: Your application can use this action to determine when the custom button in the controller (if any) is clicked.

Movie Controller Functions

This section describes one new function that is supported by movie controller components that handle movie events. A new flag for an existing function has also been added.

Handling Movie Events

QuickTime 2.1 provided two changes from movie event handling as documented in Chapter 2 of *Inside Macintosh: QuickTime Components*. First, a new flag can now be returned by the `MCGetControllerInfo` function. This flag indicates when a movie is interactive, and therefore it does not make sense to play it from end to end. Second, while it has always been possible to determine if a point is contained in a movie (using `PtInMovie`), the new `MCPtInController` function provides a way to determine if a point is in the control area of a movie.

MCGetControllerInfo

The `MCGetControllerInfo` function returns a new flag to indicate that the movie is interactive and, therefore, cannot be played from start to end. For example, because users interact with a QuickTime VR movie, it cannot be played the same way that video movies can be played.

The `someflags` parameter to the `MCGetControllerInfo` function may return the following additional flag:

```
enum {
    mcInfoMovieIsInteractive    = 1 << 10,
};
```

Flag description

`mcInfoMovieIsInteractive`
: If this flag is set to 1, the movie is interactive.

MCPtInController

The `MCPtInController` function reports whether a point is in the control area of a movie.

```
pascal ComponentResult MCPtInController (
                    MovieController mc,
                    Point thePt,
                    Boolean *inController);
```

`mc`	Specifies the movie controller for the operation. You obtain this identifier from the Component Manager's `OpenComponent` or `OpenDefaultComponent` function, or from the `NewMovieController` function.
`thePt`	Specifies the point to be checked. This point must be passed in local coordinates to the controller's window. This point is checked only against the movie controller's controls, not the movie itself.
`inController`	Returns `true` if the point is in the controller; `false` if it is not.

DISCUSSION

While you could always determine if a point is contained in a movie (using `PtInMovie`), the `MCPtInController` function allows you to determine if a point is in the control area of a movie.

C H A P T E R 7

Sequence Grabber Components

This chapter discusses new features and changes to sequence grabber components as documented in Chapter 5 of *Inside Macintosh: QuickTime Components.*

New Features of Sequence Grabber Components

Sequence grabber components now allow you to assign a specific file to each channel. This allows you to collect data into more than one file at a time, which can result in improved performance by defining the files for different channels on different devices. These destination containers are referred to as *sequence grabber outputs.* See “Working with Sequence Grabber Outputs” (page 291) for a complete discussion.

Sequence grabber components now use data handler components when writing movie data. This provides greater flexibility, especially when working with special storage devices such as networks.

As discussed in Chapter 1, “Movie Toolbox,” QuickTime 2.0 introduced timecode tracks to QuickTime movies. The sequence grabber automatically creates a timecode track if the video digitizer component contains timecode information. To support timecode tracks, the sequence grabber also provides two functions that let you identify the source information associated with video data that contains timecode information. For more information about timecodes and the timecode media handler, see Chapter 1, “Movie Toolbox.”

Improved Support for Digitizing Video in Windows

Before QuickTime 2.1, an application displaying the output of a sequence grabber video channel in a window would have to pause the sequence grabber before moving or resizing a window, before a menu was pulled down, or whenever the application was put into the background. If an application failed to take these precautions, it was possible for the digitized video to draw outside the window with which it was associated.

QuickTime 2.1 solves these problems so that an application using a sequence grabber video channel in a window no longer has to take any special precautions to ensure that the video remains within the window. As long as the application calls `SGIdle` regularly, the sequence grabber automatically takes care of pausing and unpausing the video as necessary.

Storing Captured Data in Multiple Files

In QuickTime 3, the sequence grabber allows a single capture session to store the captured data across multiple files. Each channel of a capture can be placed in a separate file. In this way, sound and video can be captured to separate files, even on separate devices. It is also possible to have a single capture session place its data on several different devices in sequence. As a result, several different devices can be used in a single capture session. This enables data capture to exceed any file size limitation imposed by a file system.

Note
The file offset parameter supports 64-bit file offsets in APIs that enable capture to multiple files. The QuickTime Movie Toolbox does not currently support 64-bit file offsets, so the high 32 bits of the offset is always 0000. ◆

Application Examples

The first step in implementing multiple sequence grabber outputs during a single capture session is to create all the sequence grabber outputs. Once the outputs have been created, they must be linked together. This is done using the new `SGSetOutputNextOutput` routine. The linked outputs are used in link order. An example of creating and linking two sequence grabber outputs is shown in Listing 7-1.

Listing 7-1 Creating and linking sequence grabber outputs

```
OSErr FSSpecToSGOutput(SeqGrabComponent theSG, FSSpec *fss,
    SGOutput *output)
{
    OSErr err;
    AliasHandle alias = nil;
    err = QTNewAlias(&fss, &alias, true);
    err = SGNewOutput(theSG, (Handle)alias, rAliasType,
        seqGrabToDisk, output);
    FSSpec fss;
    SGOutput output1, output2;

    // create an FSSpec for the first file
    FSMakeFSSpec(0, 0, "\pMacintosh HD:Movie 1", &fss);

    // create the output for the first file
    FSSpecToSGOutput(theSG, &fss, &output1)

    // create an FSSpec for the second file
    FSMakeFSSpec(0, 0, "\pMacintosh HD:Movie 2", &fss);

    //create the output for the second file
    FSSpecToSGOutput(theSG, &fss, &output2)

    // direct the movie resource to the first file
    err = SGSetDataOutput(theSG, fss, seqGrabToDisk);
    if (err) goto exit;

    // finally, link the outputs
    SGSetOutputNextOutput(theSG, output1, output2);
}
```

In this example two separate outputs are created. Once these outputs are created, they are linked together using `SGSetOutputNextOutput`. The output `output1` is used first. Once that output is full, `output2` is used.

Once outputs are created, they must be associated with the sequence grabber channels that write data to these outputs. Listing 7-2 shows how this can be accomplished. This example shows how to associate the outputs created in Listing 7-2 with both a sound and a video channel.

Listing 7-2 Associating outputs with channels

```
//associate both sound and video channels with all linked outputs
SGSetChannelOutput(theSG, soundChannel, output1);
SGSetChannelOutput(theSG, videoChannel, output1);
```

You can limit output files to a particular size by specifying the maximum number of bytes to be written to a given sequence grabber output. Listing 7-3 shows an example of setting a maximum offset of 64 KB for data written to an output.

Listing 7-3 Specifying maximum data offset for an output

```
wide maxOffset;
maxOffset.hi = 0;
//set the offset to 64K
maxOffset.lo = 64 * 1024;
SGSetOutputMaximumOffset(theSG, output1, &maxOffset);
```

Sequence Grabber Components Reference

This section describes the new constants and functions associated with sequence grabber components.

Constants

This section describes the new constants for sequence grabber components.

Flags

QuickTime 1.6.1 added a new flag to the `grabPictCurrentImage` parameter to the `SGGrabPict` function:

```
enum {
    grabPictCurrentImage    = 4
};
```

Constant description

`grabPictCurrentImage`
: Set this flag to 1 to provide the fastest possible image capture. Although this flag may fail under certain circumstances, the failure is recoverable; it just will not return a picture. You can then call `SGGrabPict` again without the flag set. This routine does not pause the current preview or grab the next frame. It just causes the currently displayed image to be captured. It's a good idea to call `SGPause` before calling `SGGrabPict` with this flag.

The `flags` parameter to the `SGSettingsDialog` function is a reserved flag and can only be set to 0. QuickTime 2.1 provided a new flag value you can use to indicate that you want to display only panels that make sense for previewing:

```
enum {
    seqGrabSettingsPreviewOnly      = 1
};
```

Constant description

`seqGrabSettingsPreviewOnly`
: Set this flag to indicate that the user will be using the dialog box provided by `SGSettingsDialog` to configure the sequence grabber for previewing only, not for recording. The `SGSettingsDialog` automatically excludes any panels that aren't necessary for preview configuring, such as video or audio compression settings. Otherwise, set the `flags` parameter to 0.

Sequence Grabber Component Functions

This section describes the new and changed functions provided by sequence grabber components.

Configuring Sequence Grabber Components

SGSetDataRef

The SGSetDataRef function specifies the destination data reference for a record operation.

```
pascal ComponentResult SGSetDataRef (
                        SeqGrabComponent s,
                        Handle dataRef,
                        OSType dataRefType,
                        long whereFlags);
```

s	Specifies the component instance that identifies your connection to the sequence grabber component. You obtain this value from the Component Manager's OpenDefaultComponent or OpenComponent function.
dataRef	Contains a handle to the information that identifies the destination container.
dataRefType	Specifies the type of data reference. If the data reference is an alias, you must set the parameter to rAliasType ('alis'), indicating that the reference is an alias.
whereFlags	Contains flags that control the record operation. You must set either the seqGrabToDisk flag or the seqGrabToMemory flag to 1 (set unused flags to 0):

seqGrabToDisk
: Instructs the sequence grabber component to write the recorded data to a QuickTime movie in the container specified by the dataRef parameter. If you set this flag to 1, the sequence grabber writes the data to the container as the data is recorded. Set this flag to 0 if you set the seqGrabToMemory flag to 1. Only one of these two flags may be set to 1.

`seqGrabToMemory`
: Instructs the sequence grabber component to store the recorded data in memory until the recording process is complete. The sequence grabber then writes the recorded data to the container specified by the `dataRef` parameter. This technique provides better performance than recording directly to the container, but limits the amount of data you can record. Set this flag to 1 to record to memory. Set this flag to 0 if you set the `seqGrabToDisk` flag to 1. Only one of these two flags may be set to 1.

`seqGrabDontUseTempMemory`
: Prevents the sequence grabber component from using temporary memory during the record operation. By default, the sequence grabber component and its channel components use as much temporary memory as necessary to perform the record operation. Set this flag to 1 to prevent the sequence grabber component and its channel components from using temporary memory.

`seqGrabAppendToFile`
: Directs the sequence grabber component to add the recorded data to the data fork of the container specified by the `dataRef` parameter. By default, the sequence grabber component deletes the container and creates a new file containing one movie and the corresponding movie resource. Set this flag to 1 to cause the sequence grabber component to append the recorded data to the data fork of the container and create a new movie resource in that file.

`seqGrabDontAddMovieResource`
: Prevents the sequence grabber component from adding the new movie resource to the container specified by the `dataRef` parameter. By default, the sequence grabber component creates a new movie resource and adds that resource to the

container. Set this flag to 1 to prevent the sequence grabber component from adding the movie resource to the container. You are then responsible for adding the resource to a file, if you so desire.

`seqGrabDontMakeMovie`
Prevents the sequence grabber component from creating a movie. By default, the sequence grabber component creates a new movie resource and adds the captured data to that movie. If you set this flag to 1, the sequence grabber still calls your data function, but does not write any data to the movie file.

`seqGrabDataProcIsInterruptSafe`
Specifies that your data function is interrupt-safe, and may be called at interrupt time. This allows the sequence grabber component to present the captured data as soon as possible. Note that not all sequence grabber channel components may use this feature. It is currently supported only by sequence grabber sound channels.

DISCUSSION

The `SGSetDataRef` function allows you to specify the destination for a record operation using a data reference, and to specify other options that govern the operation. This function is similar to the `SGSetDataOutput` function, and provides you an alternative way to specify the destination.

If you are performing a preview operation, you do not need to use the `SGSetDataRef` function.

RESULT CODES

`notEnoughMemoryToGrab`	–9403	Insufficient memory for operation
`notEnoughDiskSpaceToGrab`	–9404	Insufficient disk space for operation

SGGetDataRef

The `SGGetDataRef` function allows you to determine the data reference that is currently assigned to a sequence grabber component and the control flags that would govern a record operation.

```
pascal ComponentResult SGGetDataRef (
                        SeqGrabComponent s,
                        Handle *dataRef,
                        OSType *dataRefType,
                        long *whereFlags);
```

`s` Specifies the component instance that identifies your connection to the sequence grabber component. You obtain this value from the Component Manager's `OpenDefaultComponent` or `OpenComponent` function.

`dataRef` Contains a pointer to a handle that is to receive the information that identifies the destination container.

`dataRefType` Specifies a pointer to a field that is to receive the type of data reference.

`whereFlags` Contains a pointer to a long integer that is to receive flags that control the record operation. The following flags are defined (unused flags are set to 0):

`seqGrabToDisk`
Instructs the sequence grabber component to write the recorded data to a QuickTime movie in the container specified by the `dataRef` parameter. If this flag is set to 1, the sequence grabber writes the data to the container as the data is recorded.

`seqGrabToMemory`
Instructs the sequence grabber component to store the recorded data in memory until the

recording process is complete. The sequence grabber then writes the recorded data to the container specified by the `dataRef` parameter. This technique provides better performance than recording directly to the movie file, but limits the amount of data you can record. If this flag is set to 1, the sequence grabber component is recording to memory.

`seqGrabDontUseTempMemory`
Prevents the sequence grabber component from using temporary memory during the record operation. By default, the sequence grabber component and its channel components use as much temporary memory as necessary to perform the record operation. If this flag is set to 1, the sequence grabber component and its channel components do not use temporary memory.

`seqGrabAppendToFile`
Directs the sequence grabber component to add the recorded data to the data fork of the container specified by the `dataRef` parameter. By default, the sequence grabber component deletes the container and creates a new file containing one movie and its movie resource. If this flag is set to 1, the sequence grabber component appends the recorded data to the data fork of the container and creates a new movie resource in that file.

`seqGrabDontAddMovieResource`
Prevents the sequence grabber component from adding the new movie resource to the container specified by the `dataRef` parameter. By default, the sequence grabber component creates a new movie resource and adds that resource to the container. If this flag is set to 1, the sequence grabber component does not add the movie

resource to the container. You are then responsible for adding the resource to a file, if you so desire.

`seqGrabDontMakeMovie`
: Prevents the sequence grabber component from creating a movie. By default, the sequence grabber component creates a new movie resource and adds the captured data to that movie. If this flag is set to 1, the sequence grabber still calls your data function, but does not write any data to the container.

`seqGrabDataProcIsInterruptSafe`
: Specifies that your data function is interrupt-safe, and may be called at interrupt time. This allows the sequence grabber component to present the captured data as soon as possible. Note that not all sequence grabber channel components may use this feature.

DISCUSSION

The `SGGetDataRef` function allows you to determine the data reference that is currently assigned to a sequence grabber component and the control flags that would govern a record operation.

You set these characteristics by calling `SGSetDataRef` (page 284). If you have not set these characteristics before calling the `SGGetDataRef` function, the returned data is meaningless.

SGSettingsDialog

The `SGSettingsDialog` function has a new flag value that you can pass in the `flags` parameter. Previously, you could only set this parameter to 0. Pass the new flag, `seqGrabSettingsPreviewOnly`, to indicate that you want to display only panels that make sense for previewing. In particular, the video compression will not be displayed. Use this flag for applications that allow a live video signal to be viewed but not captured.

`flags`	Either set this to 0 or to `seqGrabSettingsPreviewOnly`. The function supports the following flag value:

`seqGrabSettingsPreviewOnly`
Use this value if you want to view but not capture a live video signal. Otherwise, set the `flags` parameter to 0.

Controlling Sequence Grabber Components

SGGrabPict

QuickTime 1.6.1 added a new flag to the `grabPictCurrentImage` parameter for the `SGGrabPict` function.

```
enum {
    grabPictCurrentImage    = 4
};
```

Constant descriptions

`grabPictCurrentImage`
Set this flag to 1 to provide the fastest possible image capture. Although this flag may fail under certain circumstances, this failure is recoverable; it just will not return a picture. You can then call `SGGrabPict` again without the flag set. This routine does not pause the current preview or grab the next frame; it just causes the currently displayed image to be captured. It's a good idea to call `SGPause` before calling `SGGrabPict` with this flag.

SGGetMode

The `SGGetMode` function returns the mode for a sequence grabber component.

```
pascal ComponentResult SGGetMode (
                        SeqGrabComponent s,
                        Boolean *previewMode,
                        Boolean *recordMode);
```

`s`	Specifies the component instance that identifies your connection to the sequence grabber component. You obtain this value from the Component Manager's `OpenDefaultComponent` or `OpenComponent` function.
`previewMode`	Contains a pointer to a Boolean. The sequence grabber component sets this field to `true` if the component is in preview mode.
`recordMode`	Contains a pointer to a Boolean. The sequence grabber component sets this field to `true` if the component is in record mode.

DISCUSSION

The `SGGetMode` function provides a convenient mechanism for determining whether a sequence grabber component is in preview mode or record mode.

Working with Sequence Grabber Outputs

In order to allow sequence grabber components to capture to more than one data reference at a time, QuickTime 2.0 introduced the concept of a sequence grabber output. A *sequence grabber output* ties a sequence grabber channel to a specified data reference for the output of captured data.

If you are capturing to a single movie file, you can continue to use the `SGSetDataOutput` function (or the new `SGSetDataRef` function) to specify the sequence grabber's destination. However, if you want to capture movie data into several different files or data references, you must use sequence grabber outputs to do so. Even if you are using outputs, you must still use the `SGSetDataOutput` function or the `SGSetDataRef` function to identify where the sequence grabber should create the movie resource.

You are responsible for creating outputs, assigning them to sequence grabber channels, and disposing of them when you are done. Sequence grabber components provide a number of functions for managing outputs:

- The `SGNewOutput` function creates a new output.
- The `SGDisposeOutput` function disposes of an output.
- The `SGSetOutputFlags` function configures the output.
- The `SGSetChannelOutput` function assigns an output to a channel.
- The `SGGetDataOutputStorageSpaceRemaining` function determines how much space is left in the output.

SGNewOutput

The `SGNewOutput` function creates a new sequence grabber output.

```
pascal ComponentResult SGNewOutput (
                     SeqGrabComponent s,
                     Handle dataRef,
                     OSType dataRefType,
                     long whereFlags,
                     SGOutput *sgOut);
```

`s`	Specifies the component instance that identifies your connection to the sequence grabber component. You obtain this value from the Component Manager's `OpenDefaultComponent` or `OpenComponent` function.
`dataRef`	Contains a handle to the information that identifies the destination container.
`dataRefType`	Specifies the type of data reference. If the data reference is an alias, you must set the parameter to `rAliasType` (`'alis'`), indicating that the reference is an alias.
`whereFlags`	Contains flags that control the record operation. You must set either the `seqGrabToDisk` flag or the `seqGrabToMemory` flag to 1 (set unused flags to 0):

`seqGrabToDisk`
: Instructs the sequence grabber component to write the recorded data to a QuickTime movie in the container specified by the `dataRef` parameter. If you set this flag to 1, the sequence grabber writes the data to the container as the data is recorded. Set this flag to 0 if you set the `seqGrabToMemory` flag to 1. Only one of these two flags may be set to 1.

`seqGrabToMemory`
: Instructs the sequence grabber component to store the recorded data in memory until the recording process is complete. The sequence grabber then writes the recorded data to the container specified by the `dataRef` parameter. This technique provides better performance than recording directly to the container, but limits the amount of data you can record. Set this flag to 1 to record to memory. Set this flag to 0 if you set the `seqGrabToDisk` flag to 1. Only one of these two flags may be set to 1.

`seqGrabDontUseTempMemory`
: Prevents the sequence grabber component from using temporary memory during the record operation. By default, the sequence grabber component and its channel components use as much temporary memory as necessary to perform the record operation. Set this flag to 1 to prevent the sequence grabber component and its channel components from using temporary memory.

`seqGrabAppendToFile`
: Directs the sequence grabber component to add the recorded data to the data fork of the container specified by the `dataRef` parameter. By default, the sequence grabber component deletes the container and creates a new file containing one movie and the corresponding movie resource. Set this flag to 1 to cause the sequence

grabber component to append the recorded data to the data fork of the container and create a new movie resource in that file.

`seqGrabDontAddMovieResource`
: Prevents the sequence grabber component from adding the new movie resource to the container specified by the `dataRef` parameter. By default, the sequence grabber component creates a new movie resource and adds that resource to the container. Set this flag to 1 to prevent the sequence grabber component from adding the movie resource to the container. You are then responsible for adding the resource to a file, if you so desire.

`seqGrabDontMakeMovie`
: Prevents the sequence grabber component from creating a movie. By default, the sequence grabber component creates a new movie resource and adds the captured data to that movie. If you set this flag to 1, the sequence grabber still calls your data function, but does not write any data to the movie file.

`seqGrabDataProcIsInterruptSafe`
: Specifies that your data function is interrupt-safe, and may be called at interrupt time. This allows the sequence grabber component to present the captured data as soon as possible. Note that not all sequence grabber channel components may use this feature.

`output` Contains a pointer to a sequence grabber output. The sequence grabber component returns an output identifier. You can then use this identifier with other sequence grabber component functions.

DISCUSSION

The `SGNewOutput` function creates a new sequence grabber output. You specify the output's destination container using a data reference. Once you have created the sequence grabber output, you can use the `SGSetChannelOutput` function to assign the output to a sequence grabber channel.

RESULT CODES

`paramErr`	–50	Invalid parameter specified

SGDisposeOutput

The `SGDisposeOutput` function disposes of an existing output.

```
pascal ComponentResult SGDisposeOutput (
                        SeqGrabComponent s,
                        SGOutput sgOut);
```

`s`	Specifies the component instance that identifies your connection to the sequence grabber component. You obtain this value from the Component Manager's `OpenDefaultComponent` or `OpenComponent` function.
`sgOut`	Identifies the sequence grabber output for this operation. You obtain this identifier by calling the `SGNewOutput` function.

DISCUSSION

You use the `SGDisposeOutput` function to dispose of an existing output. If any sequence grabber channels are using this output, the sequence grabber component assigns them to an undefined output.

Note that you cannot dispose of an output when the sequence grabber component is in record mode.

RESULT CODES

`cantDoThatInCurrentMode`	–9402	Request invalid in current mode

SGSetChannelOutput

The `SGSetChannelOutput` function assigns an output to a channel.

```
pascal ComponentResult SGSetChannelOutput (
                        SeqGrabComponent s,
                        SGChannel c,
                        SGOutput sgOut);
```

`s`	Specifies the component instance that identifies your connection to the sequence grabber component. You obtain this value from the Component Manager's `OpenDefaultComponent` or `OpenComponent` function.
`c`	Identifies the channel for this operation by passing your connection identifier. You connect to a channel component by calling the `SGNewChannel` or `SGNewChannelFromComponent` functions.
`sgOut`	Identifies the sequence grabber output for this operation. You obtain this identifier by calling the `SGNewOutput` function.

DISCUSSION

You use the `SGSetChannelOutput` function to assign an output to a channel. Note that when you call the `SGSetDataRef` or `SGSetDataOutput` functions the sequence grabber component sets every channel to the specified file or container. If you want to use different outputs, you must use this function to assign the channels appropriately. One output may be assigned to one or more channels.

RESULT CODES

`badSGChannel`	–9406	Invalid channel specified

SGSetOutputFlags

The `SGSetOutputFlags` function configures an existing sequence grabber output.

```
pascal ComponentResult SGSetOutputFlags (
                        SeqGrabComponent s,
                        SGOutput sgOut,
                        long whereFlags);
```

`s` Specifies the component instance that identifies your connection to the sequence grabber component. You obtain this value from the Component Manager's `OpenDefaultComponent` or `OpenComponent` function.

`sgOut` Identifies the sequence grabber output for this operation. You obtain this identifier by calling the `SGNewOutput` function.

`whereFlags` Contains flags that control the record operation. You must set either the `seqGrabToDisk` flag or the `seqGrabToMemory` flag to 1 (set unused flags to 0):

`seqGrabToDisk`
Instructs the sequence grabber component to write the recorded data to a QuickTime movie in the container specified by the `dataRef` parameter. If you set this flag to 1, the sequence grabber writes the data to the container as the data is recorded. Set this flag to 0 if you set the `seqGrabToMemory` flag to 1. Only one of these two flags may be set to 1.

`seqGrabToMemory`
Instructs the sequence grabber component to store the recorded data in memory until the recording process is complete. The sequence grabber then writes the recorded data to the container specified by the `dataRef` parameter. This technique provides better performance than

recording directly to the container, but limits the amount of data you can record. Set this flag to 1 to record to memory. Set this flag to 0 if you set the `seqGrabToDisk` flag to 1. Only one of these two flags may be set to 1.

`seqGrabDontUseTempMemory`
Prevents the sequence grabber component from using temporary memory during the record operation. By default, the sequence grabber component and its channel components use as much temporary memory as necessary to perform the record operation. Set this flag to 1 to prevent the sequence grabber component and its channel components from using temporary memory.

`seqGrabAppendToFile`
Directs the sequence grabber component to add the recorded data to the data fork of the container specified by the `dataRef` parameter. By default, the sequence grabber component deletes the container and creates a new file containing one movie and the corresponding movie resource. Set this flag to 1 to cause the sequence grabber component to append the recorded data to the data fork of the container and create a new movie resource in that file.

`seqGrabDontAddMovieResource`
Prevents the sequence grabber component from adding the new movie resource to the container specified by the `dataRef` parameter. By default, the sequence grabber component creates a new movie resource and adds that resource to the container. Set this flag to 1 to prevent the sequence grabber component from adding the movie resource to the container. You are then responsible for adding the resource to a file, if you so desire.

`seqGrabDontMakeMovie`
: Prevents the sequence grabber component from creating a movie. By default, the sequence grabber component creates a new movie resource and adds the captured data to that movie. If you set this flag to 1, the sequence grabber still calls your data function, but does not write any data to the movie file.

`seqGrabDataProcIsInterruptSafe`
: Specifies that your data function is interrupt-safe, and may be called at interrupt time. This allows the sequence grabber component to present the captured data as soon as possible. Note that not all sequence grabber channel components may use this feature.

DISCUSSION

The `SGSetOutputFlags` function allows you to configure an existing sequence grabber output.

RESULT CODES

`paramErr`	–50	Invalid parameter specified
`cantDoThatInCurrentMode`	–9402	Request invalid in current mode

SGGetDataOutputStorageSpaceRemaining

The `SGGetDataOutputStorageSpaceRemaining` function returns the amount of space remaining in the data reference associated with an output.

```
pascal ComponentResult SGGetDataOutputStorageSpaceRemaining (
                      SeqGrabComponent s,
                      SGOutput sgOut,
                      unsigned long *space);
```

`s`	Specifies the component instance that identifies your connection to the sequence grabber component. You obtain this value from the Component Manager's `OpenDefaultComponent` or `OpenComponent` function.
`sgOut`	Identifies the sequence grabber output for this operation. You obtain this identifier by calling the `SGNewOutput` function.
`space`	Contains a pointer to an unsigned long. The sequence grabber component returns a value that indicates the number of bytes of space remaining in the data reference associated with the output.

DISCUSSION

The `SGGetDataOutputStorageSpaceRemaining` function allows you to determine the amount of space remaining in the data reference associated with an output. Use this function in place of the `SGGetStorageSpaceRemaining` function in cases where you are working with more than one output.

RESULT CODES

`paramErr`	–50	Invalid parameter specified

Storing Captured Data in Multiple Files

SGSetOutputNextOutput

The `SGSetOutputNextOutput` function allows you to specify the order in which sequence grabber outputs should be used.

```
pascal ComponentResult SGSetOutputNextOutput(SeqGrabComponent s,
      SGOutput sgOut, SGOutput nextOut);
```

`s`	Specifies a component instance identifying your connection to the sequence grabber component. You obtain this value from the Component Manager's `OpenComponent` function.

`sgOut`	Specifies the current output to use. When a new output is created, its `nextOut` is set to `nil`.
`nextOut`	Specifies the next output to be used. To specify that this is the last output, set this value to `nil`.

DISCUSSION

The `SGSetOutputNextOutput` function should not be called while recording.

SGGetOutputNextOutput

The `SGGetOutputNextOutput` function returns the next sequence grabber output for the specified output.

```
pascal ComponentResult SGGetOutputNextOutput(SeqGrabComponent s,
      SGOutput sgOut, SGOutput *nextOut);
```

`s`	Specifies a component instance identifying your connection to the sequence grabber component. You obtain this value from the Component Manager's `OpenComponent` function.
`sgOut`	Specifies the current sequence grabber output.
`nextOut`	Contains a pointer to the next output to be used. If there is no next output, this value is `nil`.

SGSetOutputMaximumOffset

The `SGSetOutputMaximumOffset` function specifies the maximum offset for data written to a specified sequence grabber output.

```
pascal ComponentResult SGSetOutputMaximumOffset(SeqGrabComponent s,
      SGOutput sgOut, const wide *maxOffset);
```

`s`	Specifies a component instance identifying your connection to the sequence grabber component. You obtain this value from the Component Manager's `OpenComponent` function.

`sgOut`	Specifies the current sequence grabber output.
`maxOffset`	Contains a pointer to the value of the maximum offset for data written to this output.

DISCUSSION

If an attempt is made to write data beyond the maximum offset, sequence grabber switches to the next output specified by `SGSetOutputNextOutput`. If no more outputs are available, an end-of-file error is returned and recording ends.

SGGetOutputMaximumOffset

The `SGGetOutputMaximumOffset` function returns the maximum offset for data written to the specified sequence grabber output.

```
pascal ComponentResult SGGetOutputMaximumOffset(SeqGrabComponent s,
      SGOutput sgOut, wide *maxOffset);
```

`s`	Specifies a component instance identifying your connection to the sequence grabber component. You obtain this value from the Component Manager's `OpenComponent` function.
`sgOut`	Specifies the current sequence grabber output.
`maxOffset`	Contains a pointer to the value of the maximum offset for data written to this output.

DISCUSSION

The value of `SGGetOutputMaximumOffset` is initialized to (2^32 - 1) on systems with a 32-bit file system, and (2^64-1) on systems with a 64-bit file system.

SGGetOutputDataReference

The `SGGetOutputDataReference` function returns information about the data reference associated with the specified sequence grabber output.

```
pascal ComponentResult SGGetOutputDataReference(SeqGrabComponent s,
      SGOutput sgOut, Handle *dataRef, OSType *dataRefType);
```

`s`	Specifies a component instance identifying your connection to the sequence grabber component. You obtain this value from the Component Manager's `OpenComponent` function.
`sgOut`	Specifies the current sequence grabber output.
`dataRef`	A pointer to the handle in which the data reference is returned. If you do not need the data reference, set this parameter to `nil`.
`dataRefType`	A pointer to an `OSType` value in which the type of the data reference is returned. If you do not need this information, set this parameter to `nil`.

DISCUSSION

The caller is responsible for disposing of the returned handle.

Sequence Grabber Channel Components

This chapter discusses new features and changes to sequence grabber channel components as documented in Chapter 6 of *Inside Macintosh: QuickTime Components*.

New Features of Sequence Grabber Channel Components

Support for Sound Data Compression

The sound sequence grabber channel now supports compressing sound data in software when the requested format is not supported directly by the sound input driver. This feature is available only when Sound Manager 3.1 or later is installed. You can now directly capture sound in IMA and µLaw formats. Because new audio compressors and decompressor can be installed by system extensions, other audio compression formats may also be available.

Support for Sound Capture at Any Sample Rate

QuickTime 2.5 enhanced the sequence grabber sound channel to allow sound to be captured at any sample rate. The sample rate is specified, as in the past, by using `SGSetSoundInputRate`. If the requested rate is not one of the hardware rates, the sound will be captured using the closest available hardware sample rate and will be rate-converted in software to the requested rate.

In most cases, sound capture hardware does not run at the same clock rate as the motherboard crystal used to generate time stamps. Sound capture hardware also rarely runs on the same clock as video capture hardware. Over time, drift between these clocks can result in the loss of synchronization between sound and video.

QuickTime measures the drift over the duration of the capture and applies an adjustment to the sample rate of the audio to keep things synchronized. In nearly all cases, this is the right thing to do. If your hardware really knows that it always captures at the correct sample rate, it can tell QuickTime not to adjust the sample rate.

To prohibit adjustment of the sample rate, implement the `'qtrt'` resource in your sound input device's `GetInfo` routine. The argument passed is a pointer to a `short`. Set the `short` to `true` to indicate you don't want sample rate adjustment to be applied.

Working With Channel Characteristics

The sequence grabber supports two new functions, `SGChannelSetDataSourceName` and `SGChannelGetDataSourceName`, that allow you to specify the source identification information associated with a sequence grabber channel. For more information about timecodes and the timecode media handler, see Chapter 1, "Movie Toolbox."

Capturing to Multiple Files

In QuickTime 3, sequence grabber channel components can capture data into multiple files. Capturing to multiple files can improve the performance and flexibility of captures and enable larger total captures.

Creating a Sequence Grabber Component that Captures Multiple Files

You can create a sequence grabber component that can capture to multiple files by doing the following in your sequence grabber component:

- Use `SGAddExtendedMovieData` rather than `SGAddMovieData` to write data.
- In the `SGWriteSamples` routine, instead of using `SGGetNextFrameReference`, use `SGGetNextExtendedFrameReference`.

An example of how to do this is shown in Listing 8-1. This example also shows how to use the `SGAddOutputDataRefToMedia` helper routine to easily manage the multiple files in which the captured data is stored.

Listing 8-1 Channel capture and managing multiple output files

```
Track aTrack = NewMovieTrack(theMovie, width, height, 0);
Media aMedia = NewTrackMedia(aTrack, TextMediaType,
                        kMediaTimeScale, nil, 0);
SeqGrabExtendedFrameInfo fi;
SGOutput lastOutput = nil;
long i;
OSErr err;
fi.frameChannel = store->self;
i = -1;
do {
    TimeValue frameDuration;
    err = SGGetNextExtendedFrameReference(store->grabber, &fi,
            &frameDuration, &i);
            if (err) {
                if (err == paramErr)
                err = noErr;
                break;
            }
    // switch to the next data reference
    if (lastOutput != fi.frameOutput) {
        err = SGAddOutputDataRefToMedia(store->grabber,
            fi.frameOutput, aMedia, sampleDescription);
        if (err) goto exit;
        lastOutput = fi.frameOutput;
    }
    //note that only the low 32 bits of the file offset are used here
    err = AddMediaSampleReference(aMedia,
            fi.frameOffset.lo, fi.frameSize,
            frameDuration,
            sampleDescription, 1,
            0, 0);
} while (err == noErr);
exit:
    if (alias) DisposeHandle((Handle)alias);
    return err;
```

In this example, the default data reference is not defined when `NewTrackMedia` is called. Instead, the default data reference is defined by the first call to `SGAddOutputDataRefToMedia`. This approach provides added flexibility by allowing movies to be captured to data handlers other than the standard file system data handler.

Sequence Grabber Channel Components Reference

Data Types

Frame Information Structure

The frame information structure defines a frame for a sequence grabber component and its sequence grabber channel components. The `SeqGrabExtendedFrameInfo` data type defines the format of a frame information structure. `SeqGrabExtendedFrameInfo` is an extension of `SeqGrabFrameInfo`, described in *Inside Macintosh: QuickTime Components*.

```
struct SeqGrabExtendedFrameInfo {
    wide            frameOffset;
    long            frameTime;
    long            frameSize;
    SGChannel       frameChannel;
    long            frameRefCon;
    SGOutput        frameOutput;
};
typedef struct SeqGrabExtendedFrameInfo SeqGrabExtendedFrameInfo;
typedef SeqGrabExtendedFrameInfo *SeqGrabExtendedFrameInfoPtr;
```

FIELD DESCRIPTIONS

`frameOffset`	Specifies the offset to the sample. Note that this is a 64-bit value.

`frameTime`	Specifies the time at which the frame was captured by a sequence grabber channel component. The time value is relative to the data sequence. The channel component must choose a time scale and use it consistently for all sample references.
`frameSize`	Specifies the number of bytes in the current sample.
`frameChannel`	Identifies the current connection to the channel component.
`frameRefCon`	Contains a reference constant for use by the channel component. The channel component uses this constant in any appropriate way—for example, to store a reference to frame differencing information for a time-compressed sequence.
`frameOutput`	Identifies the sequence grabber output used to store captured data referenced by the current record.

DISCUSSION

`SeqGrabExtendedFrameInfo` differs from `SeqGrabFrameInfo` in two respects: `frameOffset` is a 64-bit value, and `frameOutput` does not exist in `SeqGrabFrameInfo`.

Functions

This section describes functions that can be implemented by sequence grabber channel components.

SGAddExtendedFrameReference

```
pascal ComponentResult SGAddExtendedFrameReference(SeqGrabComponent s,
      SeqGrabExtendedFrameInfoPtr frameInfo);
```

`s`	Identifies a component instance that identifies the sequence grabber component connected to your channel component. The sequence grabber component provides this value when `SGInitChannel` is called.

`frameInfo`	Contains a pointer to a frame information structure defined by the `SeqGrabExtendedFrameInfo` data type. Your component must place the appropriate information into the record identified by this parameter. The format and content of the frame information structure are described on page 8.

DISCUSSION

`SGAddExtendedFrameReference` differs from `SGAddFrameReference` in one respect: it uses `SeqGrabExtendedFrameInfoPtr` instead of a pointer to a `SeqGrabFrameInfo` structure.

SGAddExtendedMovieData

The `SGAddExtendedMovieData` function allows your channel component to add data to a movie without writing data to a movie file.

```
pascal ComponentResult SGAddExtendedMovieData(SeqGrabComponent s,
      SGChannel c, Ptr p, long len, wide *offset, long chRefCon,
      TimeValue time, short writeType, SGOutput *whichOutput);
```

`s`	Identifies a component instance that identifies the sequence grabber component connected to your channel component. The sequence grabber component provides this value when `SGInitChannel` is called.
`c`	Identifies the connection to your channel.
`p`	Specifies the location of the data to be added to the movie.
`len`	Specifies the number of bytes of data to be added to the movie.
`offset`	Contains a pointer to a wide integer that receives the offset to the new data in the movie. If the movie is in memory, the returned offset reflects the location the data will have in the movie on a permanent storage device.
`chRefCon`	Contains the reference constant for your channel.
`time`	Specifies the time at which the frame was captured, expressed in the time scale associated with your channel.

`writeType` Specifies the type of write operation to be used. The available values are:

`seqGrabWriteAppend`
Append new data to the end of the file. The sequence grabber sets the field referenced by the `offset` parameter to reflect the location at which data is added.

`seqGrabWriteReserve`
Do not write data to the output file, but reserve space in that file for the amount of data specified by the `len` parameter. The sequence grabber sets the field referenced by the `offset` parameter to the location of the reserved space.

`seqGrabWriteFill`
Write the data to the location specified by the field referenced by the offset parameter. The sequence grabber sets that field to the location of the byte following the last previously written byte. Use this option to fill space reserved when the `writeType` parameter was set to `seqGrabWriteReserve`.

`whichOutput` Contains a pointer to the sequence grabber output to which the data was written.

DISCUSSION

`SGAddExtendedMovieData` differs from `SGAddMovieData` in two respects: the `offset` parameter allows a 64-bit value, and the `whichOutput` parameter does not exist in `SGAddMovieData`.

The use of `whichOutput` depends on the value passed in the `writeType` parameter. If the writeType is `seqGrabWriteAppend` or `seqGrabWriteReserve`, the `whichOutput` parameter is a return value specifying the sequence grabber output to which data was written or in which space was reserved. If the `writeType` is `seqGrabWriteFill`, the `whichOutput` parameter is an input value indicating which sequence grabber output the data should be written to.

SGAddOutputDataRefToMedia

The `SGAddOutputDataRefToMedia` function lets you manage capture sessions that involve multiple data references. You pass to `SGAddOutputDataRefToMedia` a sequence grabber output along with a media and sample description, and `SGAddOutputDataRefToMedia` adds the data reference to the data reference list of the specified media. `SGAddOutputDataRefToMedia` also updates the data reference index field of the sample description to refer to the data reference.

```
pascal ComponentResult SGAddOutputDataRefToMedia(SeqGrabComponent s,
      SGOutput sgOut, Media theMedia, SampleDescriptionHandle desc);
```

`s`	Identifies a component instance that identifies the sequence grabber component connected to your channel component. The sequence grabber component provides this value when `SGInitChannel` is called.
`sgOut`	Contains a pointer to the current sequence grabber output.
`theMedia`	Identifies the current media. This media identifier is supplied by the Movie Toolbox.
`desc`	A handle containing an index assigned to the data by the toolbox call.

DISCUSSION

The `SGAddOutputDataRefToMedia` function is usually called from the `SGAddSamples` routine of the sequence grabber channel component.

SGChannelGetDataSourceName

The `SGChannelGetDataSourceName` function returns the data source name for a track.

```
pascal ComponentResult SGChannelGetDataSourceName (
                     SGChannel c,
                     Str255 name,
                     ScriptCode *scriptTag);
```

`c`	Identifies the channel connection for this operation.
`name`	Identifies a string that is to receive the source identification information. Set this parameter to `nil` if you do not want to retrieve the name.
`scriptTag`	Specifies a field that is to receive the source information's language code. Set this parameter to `nil` if you do not want this information.

DESCRIPTION

The `SGChannelGetDataSourceName` function allows you to get the source information specified with `SGChannelSetDataSourceName`.

SGChannelGetRequestedDataRate

The `SGChannelGetRequestedDataRate` function returns the current maximum data rate requested for a channel.

```
pascal ComponentResult SGChannelGetRequestedDataRate (
                        SGChannel c,
                        long *bytesPerSecond);
```

`c`	Identifies the channel connection for this operation.
`bytesPerSecond`	Points to a field that is to receive the maximum data rate requested by the sequence grabber component.This field is set to 0 if the sequence grabber has not set any restrictions.

DESCRIPTION

The `SGChannelGetRequestedDataRate` function allows the sequence grabber component to retrieve the current maximum data rate value from your channel component.

RESULT CODES

`badComponentSelector`	0x80008002	Function not supported

SGChannelSetDataSourceName

The `SGChannelSetDataSourceName` function sets the data source name for a track.

```
pascal ComponentResult SGChannelSetDataSourceName (
                        SGChannel c,
                        ConstStr255Param name,
                        ScriptCode scriptTag);
```

`c`	Identifies the channel connection for this operation.
`name`	Identifies a string that contains the source identification information.
`scriptTag`	Specifies the language of the source identification information.

DISCUSSION

The `SGChannelSetDataSourceName` function allows you to set the source information for a sequence grabber channel. You must set this information before you start digitizing.

This source information identifies the source of the video data (say, a videotape name). The sequence grabber channel stores this information in a timecode track in the movie created after the capture is complete. If the video digitizer does not provide timecode information, the sequence grabber does not save this information.

This function is currently supported only by video channels.

SGChannelSetRequestedDataRate

The `SGChannelSetRequestedDataRate` function specifies the maximum requested data rate for a channel.

```
pascal ComponentResult SGChannelSetRequestedDataRate (
                     SGChannel c,
                     long bytesPerSecond);
```

`c`	Identifies the channel connection for this operation.
`bytesPerSecond`	Specifies the maximum data rate requested by the sequence grabber component. The sequence grabber component sets this parameter to 0 to remove any data-rate restrictions.

DISCUSSION

The `SGChannelSetRequestedDataRate` function allows the sequence grabber component to specify the maximum rate at which it would like to receive data from your channel component.

The data rate supplied by the sequence grabber component represents a requested data rate. Your component may not be able to observe that rate under all conditions.

RESULT CODES

`badComponentSelector`	0x80008002	Function not supported

SGGetAdditionalSoundRates

The `SGGetAdditionalSoundRates` function returns the additional sound sample rates added to the specified sequence grabber sound channel.

```
pascal ComponentResult SGGetAdditionalSoundRates(SGChannel c,
                     Handle *rates);
```

`c`	Identifies the channel connection for this operation.

`rates`	A pointer to handle where the list of additional sample rates should be returned.

DISCUSSION

`SGGetAdditionalSoundRates` returns a copy of the list of additional samples rates passed to the `SSGetAdditionalSoundRates` previously. If no additional sample rates have been set, `SGGetAdditionalSoundRates` sets the rates handle to `nil`. The caller of this routine is responsible for disposing of the returned rates handle.

SGGetNextExtendedFrameReference

The `SGGetNextExtendedFrameReference` function allows a channel component to retrieve the sample references stored previously by `SGAddExtendedMovieData` or `SGAddExtendedFrameReference`.

```
pascal ComponentResult SGGetNextExtendedFrameReference(SeqGrabComponent
      s, SeqGrabExtendedFrameInfoPtr frameInfo, TimeValue *frameDuration,
      long *frameNumber);
```

`s`	Identifies a component instance that identifies the sequence grabber component connected to your channel component. The sequence grabber component provides this value when `SGInitChannel` is called.
`frameInfo`	Contains a pointer to a frame information structure defined by the `SeqGrabExtendedFrameInfo` data type. Your component must place the appropriate information into the record identified by this parameter. The format and content of the frame information structure are described on page 8.
`frameDuration`	Contains a pointer to a time value. The sequence grabber component calculates the duration of the specified frame and returns that duration in the structure this parameter refers to. The sequence grabber component cannot calculate the duration of the last frame in a sequence. For the last frame, the time value is set to –1.

`frameNumber`	Contains a pointer to a long integer representing the frame number. Frame numbers may not be sequential, and may not start at 0. To retrieve information about the first frame in a movie, set the field referred to by the `frameNumber` parameter to –1.

DISCUSSION

`SGGetNextExtendedFrameReference` differs from `SGGetNextFrameReference` in that it fills out a `SeqGrabExtendedFrameInfo` structure instead of a `SeqGrabFrameInfo` structure.

Your channel component can process frame references sequentially or randomly. You can specify any relative frame for which you want to retrieve information.

RESULT CODES

`paramErr`	`-50`	Invalid parameter specified

SGGetPreferredPacketSize

The `SGGetPreferredPacketSize` function returns the preferred packet size for the sequence grabber component.

```
pascal ComponentResult SGGetPreferredPacketSize (
                        SGChannel c,
                        long *preferredPacketSizeInBytes);
```

`c`	Identifies the channel connection for this operation.
`preferredPacketSizeInBytes`	The preferred packet size in bytes.

DISCUSSION

The `SGGetPreferredPacketSize` function was added in QuickTime 2.5 to support video conferencing applications.

SGGetUserVideoCompressorList

The `SGGetUserVideoCompressorList` function returns the video compression formats to be displayed by the specified sequence grabber video channel.

```
pascal ComponentResult SGGetUserVideoCompressorList(SGChannel c,
                        Handle *compressorTypes);
```

`c`	Identifies the channel connection for this operation.
`compressorTypes`	A pointer to handle where the list of video compression formats should be returned.

DISCUSSION

`SGGetUserVideoCompressorList` returns a copy of the list of video compression formats passed to the `SGSetUserVideoCompressorList` previously. If no video compression formats have been set, `SGGetUserVideoCompressorList` sets the compressorTypes handle to `nil`. The caller of this routine is responsible for disposing of the returned video compression formats handle.

SGSetAdditionalSoundRates

The `SGSetAdditionalSoundRates` function allows an application to specify a list of sound sample rates to be included in the sequence grabber's sound settings dialog box. If any of the requested rates are not supported directly by the available sound capture hardware, sound will be captured at one of the available hardware rates and then rate converted in software to the requested rate.

```
pascal ComponentResult SGSetAdditionalSoundRates(SGChannel c,
                        Handle rates);
```

`c`	Identifies the channel connection for this operation.
`rates`	A handle containing a list of unsigned 32-bit fixed-point values. The sequence grabber channel determines the number of sample rates contained in the handle, based on the size of the handle.

DISCUSSION

The sequence grabber channel makes a copy of the additional rates handle. Therefore, your application can immediately dispose of the additional rates handle after making this call.

SGSetPreferredPacketSize

The `SGSetPreferredPacketSize` function sets the preferred packet size for the sequence grabber channel component.

```
pascal ComponentResult SGSetPreferredPacketSize (
                      SGChannel c,
                      long preferredPacketSizeInBytes);
```

`c`	Identifies the channel connection for this operation.
`preferredPacketSizeInBytes`	The preferred packet size in bytes.

DISCUSSION

The `SGSetPreferredPacketSize` function was added in QuickTime 2.5 to support video conferencing applications.

SGSetUserVideoCompressorList

The `SGSetUserVideoCompressorList` function allows an application to specify the list of video compression formats to be included in the sequence grabber's video settings dialog box. This allows an application to limit the number of video compression formats that will be displayed to the user. For applications using the sequence grabber for a very specific purpose, this allows inappropriate compression choices to be filtered out.

```
pascal ComponentResult SGSetUserVideoCompressorList(SGChannel c,
                      Handle compressorTypes);
```

`c`	Identifies the channel connection for this operation.
`compressorTypes`	A handle containing a list of `OSTypes` indicating which video compression formats should be displayed. The sequence grabber channel determines the number of video compression formats contained in the handle based on the size of the handle.

DISCUSSION

The sequence grabber channel makes a copy of the video compression formats handle. Therefore, your application can immediately dispose of the video compression formats handle after making this call.

SGWriteExtendedMovieData

The `SGWriteExtendedMovieData` function allows your channel component to add data to a movie.

```
pascal ComponentResult SGWriteExtendedMovieData(SeqGrabComponent s,
      SGChannel c, Ptr p, long len, wide *offset, SGOutput *sgOut);
```

`s`	Identifies a component instance that identifies the sequence grabber component connected to your channel component. The sequence grabber component provides this value when `SGInitChannel` is called.
`c`	Identifies the connection to your channel.
`p`	Specifies the location of the data to be added to the movie.
`len`	Specifies the number of bytes of data to be added to the movie.
`offset`	Contains a pointer to a wide integer that receives the offset to the new data in the movie. If the movie is in memory, the returned offset reflects the location the data will have in the movie on a permanent storage device.
`sgOut`	Contains a pointer to the sequence grabber output to which the data was written.

DISCUSSION

`SGWriteExtendedMovieData` differs from `SGWriteMovieData`, in two respects: the `offset` parameter has a 64-bit value, and the `sgOut` parameter does not exist in `SGWriteMovieData`.

Video Digitizer Components

This chapter discusses changes to video digitizer components as documented in Chapter 8 of *Inside Macintosh: QuickTime Components*. This chapter describes the new and changed constants, data types, and functions provided by these components.

New Features of Video Digitizer Components

As discussed in Chapter 1, “Movie Toolbox,” QuickTime 2.0 introduced timecode tracks to QuickTime movies. Video digitizers may now return timecode information for an incoming video signal by responding to `VDGetTimeCode` (page 328). For more information about timecodes and the timecode media handler, see Chapter 1, “Movie Toolbox.””

Video Digitizer Components Reference

Constants

Input Formats

You use the `VDGetInputFormat` function to find out the video format employed by a specified input. QuickTime defines one new constant that you can use for video digitizers that support a tuner input:

```
enum {
    tvTunerIn           = 6
};
```

Constant description

`tvTunerIn`
: The input video signal is from the television tuner connection.

Video Digitizer Component Functions

Controlling Compressed Source Devices

In QuickTime 1.5, video digitizers could provide compressed data directly to clients. However, there was no way to preflight the settings for compression. In QuickTime 2.1, a new function, `VDGetCompressionTime`, has been added to allow the video digitizer to quantify the compression time for the actual quality levels that will be used.

VDGetCompressionTime

Your component receives the `VDGetCompressionTime` request whenever a client of the digitizer wants to confirm or quantify its compression settings.

```
pascal VideoDigitizerError VDGetCompressionTime (
                     VideoDigitizerComponent ci,
                     OSType compressionType,
                     short depth,
                     Rect *srcRect,
                     CodecQ *spatialQuality,
                     CodecQ *temporalQuality,
                     unsigned long *compressTime);
```

`ci` Specifies the video digitizer component for the request. Applications obtain this reference from the Component Manager's `OpenComponent` function.

`compressionType`
: Specifies a compressor type. This value corresponds to the component subtype of the compressor component. See the chapter “Image Compression Manager” in *Inside Macintosh: QuickTime* for more information about compressor types, including valid values for this parameter.

`depth`
: Specifies the depth at which the image is to be compressed. Values of 1, 2, 4, 8, 16, 24, and 32 indicate the number of bits per pixel for color images. Values of 33, 34, 36, and 40 indicate 1-bit, 2-bit, 4-bit, and 8-bit grayscale, respectively, for grayscale images.

`srcRect`
: Contains a pointer to a rectangle defining the portion of the source image to compress.

`spatialQuality`
: Contains a pointer to a field containing the desired compressed image quality. The compressor sets this field to the closest actual quality that it can achieve. See the chapter “Image Compression Manager” in *Inside Macintosh: QuickTime* for valid values. A value of `nil` indicates that the client does not want this information.

`temporalQuality`
: Contains a pointer to a field containing the desired sequence temporal quality. The compressor sets this field to the closest actual quality that it can achieve. See the chapter “Image Compression Manager” in *Inside Macintosh: QuickTime* for valid values. A value of `nil` indicates that the client does not want this information.

`compressTime`
: Contains a pointer to a field to receive the compression time, in milliseconds. If your component cannot determine the amount of time required to compress the image, set this field to 0. A value of `nil` indicates that the client does not want this information.

DISCUSSION

The sequence grabber’s video compression settings dialog box uses this function to snap the quality slider to the correct value when working with a compression type that is specified by the video digitizer.

Your component should return a long integer indicating the maximum number of milliseconds it would require to compress the specified image.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified

Controlling Digitization

This section describes one new video digitizer component function, `VDSetDataRate`, that instructs your video digitizer component to observe a specified rate of data delivery.

VDSetDataRate

The `VDSetDataRate` function instructs your video digitizer component to limit the rate at which it delivers compressed, digitized video data.

```
pascal VideoDigitizerError VDSetDataRate (
                    VideoDigitizerComponent ci,
                    long bytesPerSecond);
```

`ci` Specifies the video digitizer component for the request. Applications obtain this reference from the Component Manager’s `OpenComponent` function.

`bytesPerSecond`
Specifies the maximum data rate requested by the application. This parameter is set to 0 to remove any data-rate restrictions.

DISCUSSION

This function is valid only for video digitizer components that can deliver compressed video (that is, components that support the `VDCompressOneFrameAsync` function). Components that support data-rate limiting set the `codecInfoDoesRateConstrain` flag to 1 in the `compressFlags` field of the

`VDCompressionList` structure returned by the component in response to the `VDGetCompressionTypes` function.

Your video digitizer component should return this data-rate limit in the `bytesPerSecond` parameter of the existing `VDGetDataRate` function.

Controlling Packet Size

VDSetPreferredPacketSize

The `VDSetPreferredPacketSize` function sets the preferred packet size for video digitizing.

```
pascal VideoDigitizerError VDSetPreferredPacketSize(
                        VideoDigitizerComponent ci,
                        long preferredPacketSizeInBytes);
```

`ci`	Specifies the video digitizer component for the request. Applications obtain this reference from the Component Manager's `OpenComponent` function.
`preferredPacketSizeInBytes`	The preferred packet size in bytes.

DESCRIPTION

This function was added in QuickTime 2.5 to support video conferencing applications.

Utility Functions

This section describes two new utility functions that may be supported by some video digitizer components.

The `VDGetTimeCode` function allows an application to retrieve timecode information.

The VDGetSoundInputSource function allows an application to retrieve information about a digitizer's sound input source.

VDGetTimeCode

The VDGetTimeCode function instructs your video digitizer component to return timecode information for the incoming video signal.

```
pascal VideoDigitizerError VDGetTimeCode (
                     VideoDigitizerComponent ci,
                     TimeRecord *atTime,
                     void *timeCodeFormat,
                     void *timeCodeTime);
```

ci	Specifies the video digitizer component for the request. Applications obtain this reference from the Component Manager's OpenComponent function.
atTime	Specifies a location to receive the QuickTime movie time value corresponding to the timecode information.
timeCodeFormat	Contains a pointer to a timecode definition structure. Your video digitizer component returns the movie's timecode definition information.
timeCodeTime	Contains a pointer to a timecode record structure. Your video digitizer component returns the time value corresponding to the movie time contained in the atTime parameter.

DISCUSSION

Typically, this function is called once, at the beginning of a capture session. The use of VDGetTimeCode assumes that the timecoding for the entire capture session will be continuous.

For more information about the timecode data structures, see Chapter 1, "Movie Toolbox."

VDGetSoundInputSource

The `VDGetSoundInputSource` function instructs your video digitizer component to return the sound input source associated with a particular video input.

```
pascal VideoDigitizerError VDGetSoundInputSource (
                       VideoDigitizerComponent ci,
                       long videoInput,
                       long *soundInput);
```

`ci`	Specifies the video digitizer component for the request. Applications obtain this reference from the Component Manager's `OpenComponent` function.
`videoInput`	Specifies the input video source for this request. Video digitizer components number video sources sequentially, starting at 0. So, to request information about the first video source, an application sets this parameter to 0. Applications can get the number of video sources supported by a video digitizer component by calling the `VDGetNumberOfInputs` function.
`soundInput`	The sound input index to use with the sound input driver returned by `VDGetSoundInputDriver`.

DISCUSSION

Some video digitizers may associate different sound inputs with each video input. The `VDGetSoundInputDriver` function returns the name of the sound input driver that the sound input is associated with.

C H A P T E R 1 0

Text Channel Component

This chapter discusses the text sequence grabber channel component and the associated text digitizer components introduced in QuickTime 2.5.

Just as video digitizer components obtain digitized video from an analog video source, text digitizer components obtain text from an external source. A text channel component is a sequence grabber channel component. A text digitizer is a new kind of component. The text channel component uses the services of text digitizer components.

For more information about sequence grabber components, see the chapter “Sequence Grabber Components” in *Inside Macintosh: QuickTime Components*. For more information about sequence grabber channel components, see the chapter “Sequence Grabber Channel Components” in *Inside Macintosh: QuickTime Components*.

About the Text Channel Component

The QuickTime text channel component allows an application to obtain text from an external source. Once obtained, this text can be previewed or recorded into a QuickTime movie. The source of the text is unknown to the text channel component; a text digitizer component (`'tdig'`) is responsible for acquiring the text from the external source. The text channel component is provided by QuickTime.

Text digitizers are separate components; they are the mechanism for presenting new sources of text data to QuickTime. Several text digitizer components are available, including one that captures closed-captioned data using an Apple TV Tuner card. For more information on creating a component, see the chapter “Component Manager” in *Mac OS For QuickTime Programmers*.

To retrieve text for previewing or for recording in a QuickTime movie, the application uses the text channel the same way in which it would use a video channel. The application calls a sequence grabber component, which, in turn, calls the text channel component. The text channel component calls the appropriate text digitizer component to retrieve the text. For more information on how to use sequence grabber components, see the chapter "Sequence Grabber Components" in *Inside Macintosh: QuickTime Components*.

Once text has been retrieved, the application can request that the sequence grabber component store the text in a text track of a QuickTime movie. For more information on the text media handler, see chapter "Movie Toolbox" in *Inside Macintosh: QuickTime*.

The QuickTime text channel component supports some, but not all, functions defined for sequence grabber channel components and sequence grabber panel components. The supported functions are described in Table 10-1.

In addition, the text channel component provides new functions implemented specifically for text; you use these functions to format captured text to be viewed or added to a text track of a movie. The new functions are described in "Text Channel Component Reference" (page 334).

Table 10-1 Functions supported by the text channel component

Usage	Supported
General sequence grabber component functions	`SGSetGWorld`
	`SGNewChannel`
	`SGStartPreview`
	`SGStartRecord`
	`SGIdle`
	`SGStop`
	`SGPause`
	`SGPrepare`
	`SGRelease`
	`SGGetChannelDeviceList`

Table 10-1 Functions supported by the text channel component (continued)

Usage	Supported
	`SGUpdate`
Functions for getting and setting channel characteristics	`SGSetChannelUsage`
	`SGGetChannelUsage`
	`SGSetChannelBounds`
	`SGGetChannelBounds`
	`SGGetChannelInfo`
	`SGSetChannelClip`
	`SGGetChannelClip`
	`SGGetChannelSampleDescription`
	`SGSetChannelDevice`
	`SGSetChannelMatrix`
	`SGGetChannelMatrix`
	`SGGetChannelTimeScale`
Text channel component functions called by sequence grabber components	`SGInitChannel`
	`SGWriteSamples`
	`SGGetDataRate`
Sequence grabber panel component functions	`SGPanelGetDitl`
	`SGPanelInstall`
	`SGPanelEvent`
	`SGPanelRemove`
	`SGPanelGetSettings`
	`SGPanelSetSettings`
	`SGPanelItem`

Text Channel Component Reference

This section describes the functions provided by the text channel component for formatting text to be previewed or added to a text track of a movie.

SGSetFontName

The `SGSetFontName` function sets the name of the font to be used to display text.

```
pascal ComponentResult SGSetFontName (
                        SGChannel c,
                        StringPtr pstr);
```

`c`	Specifies the channel for this operation.
`pstr`	A pointer to a Pascal string containing the name of the font.

DISCUSSION

You call this function to specify a font for the text channel component. If the specified font is available on the system, the text channel uses the font to display text. If the specified font is not available, the text channel uses the default system font. For more information about fonts, see *Inside Macintosh: Text*.

RESULT CODES

Component Manager errors, as documented in *Mac OS For QuickTime Programmers*.

SGSetFontSize

The `SGSetFontSize` function sets the font size to be used to display text.

```
pascal ComponentResult SGSetFontSize (
                        SGChannel c,
                        short fontSize);
```

`c`	Specifies the channel for this operation.
`fontSize`	Specifies the point size of the font.

DISCUSSION

You call this function to specify a text point size for the text channel component. The specified point size must be a positive integer value. For more information about fonts and point size, see *Inside Macintosh: Text*.

RESULT CODES

Component Manager errors, as documented in *Mac OS For QuickTime Programmers*.

SGSetTextForeColor

The `SGSetTextForeColor` function sets the color to be used to display text.

```
pascal ComponentResult SGSetTextForeColor (
                        SGChannel c,
                        RGBColor *theColor);
```

`c`	Specifies the channel for this operation.
`theColor`	Contains a pointer to an `RGBColor` structure that contains the new text color.

DISCUSSION

You call this function to set the text color for a text track.

RESULT CODES

Component Manager errors, as documented in *Mac OS For QuickTime Programmers*.

SGSetTextBackColor

The `SGSetTextBackColor` function sets the background color to be used for the text box.

```
pascal ComponentResult SGSetTextBackColor (
                     SGChannel c,
                     RGBColor *theColor);
```

`c`	Specifies the channel for this operation.
`theColor`	Contains a pointer to an `RGBColor` structure that contains the new background color.

DISCUSSION

You call this function to set the background color of a text track. The text channel component uses the specified color as the background of the text box.

RESULT CODES

Component Manager errors, as documented in *Mac OS For QuickTime Programmers*.

SGSetJustification

The `SGSetJustification` function sets the alignment to be used to display text.

```
pascal ComponentResult SGSetJustification (
                     SGChannel c,
                     short just);
```

`c`	Specifies the channel for this operation.
`just`	Specifies a constant that represents the text alignment. Possible values are `teFlushDefault`, `teCenter`, `teFlushRight`, and `teFlushLeft`.

DISCUSSION

You call this function, passing a text justification constant, to specify the alignment to be used for text in a text track. The text channel component justifies text relative to the boundaries of its text box. For more information on text alignment and the text justification constants, see the “TextEdit” chapter of *Inside Macintosh: Text*.

RESULT CODES

Component Manager errors, as documented in *Mac OS For QuickTime Programmers*.

SGGetTextRetToSpaceValue

The `SGGetTextRetToSpaceValue` function indicates whether the text channel component should replace return characters with spaces.

```
pascal ComponentResult SGGetTextRetToSpaceValue (
                        SGChannel c,
                        short *rettospace);
```

`c`	Specifies the channel for this operation.
`rettospace`	Contains a pointer to a 16-bit integer. On return, this parameter is `true` if the text channel is replacing return characters with spaces, or `false` if the text channel is not replacing return characters with spaces.

DISCUSSION

When you capture text from a closed-caption television source, the text is composed of four lines of text of up to 32 characters each, each line separated by a return character. Sometimes it is useful to replace the return characters with spaces. You can call this function to determine whether the text channel component is replacing return characters with spaces.

RESULT CODES

Component Manager errors, as documented in *Mac OS For QuickTime Programmers*.

SGSetTextRetToSpaceValue

The `SGSetTextRetToSpaceValue` function sets whether the text channel component should replace return characters with spaces.

```
pascal ComponentResult SGSetTextRetToSpaceValue (
                        SGChannel c,
                        short rettospace);
```

`c`	Specifies the channel for this operation.
`rettospace`	Specifies whether return characters should be replaced by spaces. Set this parameter to `true` if the text channel should replace return characters with spaces, or `false` if the text channel should not replace return characters with spaces.

DISCUSSION

When you capture text from a closed-caption television source, the text is composed of four lines of text of up to 32 characters each, each line separated by a return character. You call this function to request that the text channel component replace the return characters with spaces.

RESULT CODES

Component Manager errors, as documented in *Mac OS For QuickTime Programmers*.

Movie Data Exchange Components

This chapter discusses new features in movie data exchange components. For more information about movie data exchange components, see Chapter 9 of *Inside Macintosh: QuickTime Components*.

New Features of Movie Data Exchange Components

QuickTime 3 now makes it possible for your application to retrieve and store the settings of import and export components without having to present the user with a user interface, such as a settings dialog box, to accomplish the task.

Saving and Restoring Settings

QuickTime has always provided many support mechanisms for importing media from other formats into QuickTime movies, as well as exporting from QuickTime movies to other media formats. Importing and exporting are handled by movie data exchange components.

You write import and export components to allow a user to perform importing and exporting, respectively. Your component provides a routine that presents a dialog box for the user to change options. For an import component, you need to implement `MovieImportDoUserDialog`; for an export component, you must implement `MovieExportDoUserDialog`. For example, the text import component presents a dialog box with options for setting the font, size, and style of the text media it will add to the movie. The WAVE audio export component presents the standard sound compression dialog box, so that sample rate and sample size can be specified for the generated WAVE file.

In the past, when a user made a change to the configuration of a component in a settings dialog box, those changes would be lost when the component was closed. With QuickTime 3, it is now possible to retrieve the current settings from the still-open import or export component. In addition, you can restore a component's current settings to previously-retrieved settings. The restoration does not involve any user interface. This may be advantageous for application developers who want to provide preferences for the last settings used or want to perform batch importing or exporting, using previously-established settings.

For information about using the new save-and-restore component settings mechanism, refer to the section "Implementing Movie Data Exchange Components" (page 361).

Standard Compression Components and Settings

QuickTime 3 adds two new settings-related component calls to both the video and sound Standard Compression components: `SCGetSettingsAsAtomContainer` and `SCGetSettingsAsAtomContainer` These may also be useful for implementing movie data exchange components. The `SCGetSettingsAsAtomContainer` routine returns a QT atom container with the current configuration of the particular compression component. `SCSetSettingsFromAtomContainer` resets the compression component's current configuration. Applications that want to save settings for standard compression components should use these new calls.

For more information about these calls, refer to Chapter 20, "Standard Sound Dialog Component."

Exporting Text

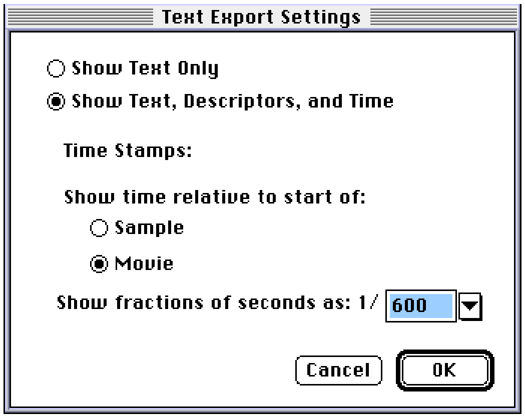
The text export and import components provide new features that make it easier to work with the data in a text track in a QuickTime movie. **Text descriptors** are formatting commands that you can embed within a text file. **Time stamps** describe a text sample's starting time and duration.

The text export and import components now make it easier to edit and format text using an external tool, such as a text editor or word processor. When you export text from a text track, you can optionally export text descriptors and time stamps for the text. You can open the text file in a word processor and make changes to the text, style, color, and time stamps. You can then import the edited text to a text track where all the timing, style, color and time stamp information will be present.

When you export text, you control whether text descriptors and time stamps are to be exported by selecting the appropriate options in the text export settings dialog box, shown in Figure 11-1. To display this dialog box programmatically, you call the `MovieExportDoUserDialog` function, described in *Inside Macintosh: QuickTime Components*.

Based on the options you specify in the text export settings dialog box, the text export component is assigned one of three text export option constants: `kMovieExportTextOnly`, `kMovieExportAbsoluteTime`, or `kMovieExportRelativeTime`.

Figure 11-1 Text export settings dialog box


Text Export Settings

Show Text Only

[✓ Show Text, Descriptors, and Time]

Time Stamps:

Show time relative to start of:

Sample

[✓ Movie]

Show fractions of seconds as: 1/ 600

If you choose "Show Text Only," the text component is assigned the export option constant `kMovieExportTextOnly`. In this case, the text component exports only text samples, without text descriptors or time stamps. This option is useful when you want to export only the text from a movie and you do not intend to import the text back into a movie.

If you select "Show Text, Descriptors, and Time," the text component is assigned one of two export option constants, depending on the format you specify for time stamps:

- If you specify time stamps to be relative to the start of the movie, the text component is assigned the export option constant `kMovieExportAbsoluteTime`. In this case, time stamps are calculated relative to the start of the movie. For example, in exported text with absolute time stamps, the time stamp `[00:00:04.000]` indicates that a text sample begins 4 seconds after the start of the movie.

- If you specify time stamps to be relative to the sample, the text component is assigned the export option constant `kMovieExportRelativeTime`. In this case, the time stamp for each sample is calculated relative to the end of the previous sample. For example, in exported text with relative time stamps, the time stamp `[00:00:04.000]` indicates that a text sample begins 4 seconds after the beginning of the previous sample. In other words, the previous sample lasts 4 seconds.

In both cases, text export component exports text, along with both text descriptors and time stamps. For more information about time stamps, see "Time Stamps" (page 350).

The text export component provides two functions you can use to access the component's text export option programmatically. To retrieve the current value of the text export option, you call `TextExportGetSettings` (page 377). To set the value of the text export option, you call `TextExportSetSettings` (page 378).

The Text Export Settings dialog box also allows you to specify the time scale the text component uses to specify the fractional part of a time stamp. The value should be between 1 and 10000, inclusive. The text export component provides two functions you can use to access the component's time scale programmatically. To retrieve the time scale, call `TextExportGetTimeFraction` (page 376). To set the time scale, call `TextExportSetTimeFraction` (page 377).

Text Descriptors

A text descriptor is a formatting command that describes the text that follows it. Exporting text with text descriptors allows you to edit text from a text track, including its formatting, in an external program, such as a text editor or word processor. When you import the edited text, the formatting you specified with the text descriptors is preserved. This provides an easy way to localize movies for different languages, correct spelling, change styles, or modify text behavior.

A text descriptor has the format`{`*descriptor*`}`. For example, the text descriptor `{bold}` sets the text style in the current text sample and all subsequent text samples. Some text descriptors, such as `{bold}`, have no parameters. Other text descriptors have one or more parameters. For text descriptors with parameters, the descriptor is followed by a colon and its parameters, separated by commas. You can specify text descriptors using either uppercase or lowercase characters, with or without spaces separating the parameters:

`{`*descriptor*: *parameter1, ..., parameterN*`}`

For example, the text descriptor `{font:New York}` sets the text font in the current text sample and all subsequent text samples to the New York font. The New York font is applied to all text until a second `{font:}` text descriptor is issued.

A text stream that contains text descriptors and time stamps should always begin with the text descriptor `{QTtext}`, followed by any number of text descriptors in any order. If the text import component detects a typographical error inside a descriptor while importing a text file, it may generate partial results or an error message stating that the text file cannot be converted.

When text with text descriptors is imported into a track, the information represented by the descriptors is stored in a text display data structure (type `TextDisplayData`). Text descriptors whose possible values are `on` and `off` are used to set flags in the `displayFlags` field of the text display data structure. Each sample in the text track has a corresponding text display data structure that contains the text attributes of the sample. For more information, see "Text Display Data Structure" (page 373).

Listing 11-1 shows a simple example of text that has been exported with text descriptors and time stamps. The text sample displays the text "I ♥ QuickTime" (the "© "character is a "♥" in the Macintosh Symbol font). The background color is black and the text color is white, except for the "♥" character, which is drawn in red.

Listing 11-1 Formatting text using text descriptors

```
{QTtext} {font:New York} {plain} {size:36} {textColor: 65535, 65535,
65535} {backColor: 0, 0, 0} {justify:center} {timeScale:600} {width:320}
{height:0} {timeStamps:absolute}
[00:00:00.000]
I {font:Symbol}{size:46}{textColor:65535, 0, 0}© {textColor: 65535,
65535, 65535}{font:New York} {plain} {size:36} QuickTime
[00:00:05.300]
```

The `{karaoke:}` text descriptor allows you to highlight groups of characters in a text sample at specified times. For example, you might want to highlight the words in a song while playing the song's sound track. The parameters for the `{karaoke:}` text descriptor are specified as a set of tuples, separated by semicolons. The set of tuples is preceded by the total number of tuples:

`{karaoke:` *count*`;` *a1, a2, a3; b1, b2, b4; ... n1, n2, n3*`}`*text sample*

Each tuple is composed of a time value (which is greater than the one specified and less then the next time value), a starting offset into the text, and an ending offset into the text; at the specified time value, the text from the starting offset to the ending offset is highlighted.

The following example shows a `{karaoke:}` text descriptor followed by some text. The time scale of the movie was set to 600. The `{karaoke:}` text descriptor has 14 tuples. The second tuple indicates that, between 125/600 seconds and 250/600 seconds, the text from offset 0 to offset 4 ("`Thun`") should be highlighted. The third tuple indicates that, after 250/600 seconds and before 350/600 seconds, the text from offset 4 to offset 7 ("`der`") should be highlighted.

```
{karaoke: 14; 0, 0, 0; 125, 0, 4; 250, 4, 7; 375, 7, 11; 500, 12, 15;
750, 16, 21; 1000, 21, 25; 1125, 25, 28; 1250, 28, 30; 1375, 31, 33;
1500, 33, 35; 1750, 36, 42; 2000, 42, 47; 2250, 48, 50;}Thunderbolt and
lightning very very fright'ning me
```

The text export and import components support the following text descriptors:

General

`{QTtext}`	Required at the beginning of any text file that contains descriptors or time stamps. If the text import component does not detect this descriptor at the beginning of the file, it assumes the file is a standard text file and will not look for descriptors or time stamps.
`{language:}`	Specifies the language of the text track. This text descriptor takes one parameter, the ordinal value of the language. For example, `{language:11}` specifies that the language of the track is Japanese. For more information on language ordinal values, see the chapter "Movie Resource Formats" in *Inside Macintosh: QuickTime.*

Text styles

`{font:}`	Specifies the name of the font to be used. For example, the text descriptor `{font:Apple Chancery}` changes the font to Apple Chancery.
`{size:}`	Specifies the point size of the text. For example, the text descriptor `{size:18}` sets the text point size to 18 points.

`{plain}`	Resets the text style. This text descriptor resets the following text descriptors: `{bold}`, `{italic}`, `{underline}`, `{shadow}`, `{outline}`, `{extend}`, and `{condense}`.
`{bold}`	Specifies bold text.
`{italic}`	Specifies italic text.
`{underline}`	Specifies underlined text.
`{outline}`	Specifies outlined text.
`{shadow}`	Specifies shadow text.
`{condense}`	Specifies condensed text.
`{extend}`	Specifies extended text.

Text dimensions

`{height:}`	Specifies the height of the text track in pixels. For example, the text descriptor `{height:50}` sets the text track height to 50 pixels. The text descriptor `{height:0}` sets the height to the best height for the text.
`{width:}`	Specifies the width of the text track in pixels. For example, the text descriptor `{width:50}` sets the text track width to 50 pixels. The text descriptor `{width:0}` sets the width to an appropriate default value or, if importing a movie, to the width of the movie.
`{textBox:}`	Specifies the coordinates of the text box. This text descriptor takes four parameters: top, left, bottom, and right. For example, if you specify {textBox:0, 0, 80, 320} (which is top, left, bottom, right), the text box originates at (0,0) and is 320 pixels wide and 80 pixels high.
`{doNotAutoScale:}`	Specifies whether to automatically scale the text if the track bounds increase. This text descriptor takes one parameter. Possible values are `on` and `off`; the default value is `off`. For example, the text descriptor `{doNotAutoScale:off}` enables auto-scaling. This corresponds to the `dfDontAutoScale` display flag.
`{clipToTextBox:}`	Specifies whether to clip to the text box. This is useful if the text overlays the video. This text descriptor takes one parameter. Possible values are `on` and `off`; the default value is `off`. This corresponds to the `dfClipToTextBox` display flag.
`{shrinkTextBox:}`	Specifies whether to recalculate the size of the text box specified by the `{textBox:}` text descriptor to fit the

dimensions of the text. If so, the new rectangle is stored with the text data. This text descriptor takes one parameter. Possible values are `on` and `off`. The value of this text descriptor is used to set a flag in the `displayFlags` field of the text display data structure. This corresponds to the `dfShrinkTextBoxToFit` display flag.

Drawing text

`{doNotDisplay:}`	Specifies whether to display the sample. This text descriptor takes one parameter. Possible values are `on` and `off`; the default value is `off`. For example, the text descriptor `{doNotDisplay:on}` causes the sample to not be displayed. This corresponds to the `dfDontDisplay` display flag.
`{justify:}`	Specifies the alignment of the text in the text box. This text descriptor takes one parameter. Possible values are `left`, `right`, `center`, and `default`. For example, the text descriptor `{justify:left}` aligns text on the left. For more information on text alignment, see the chapter “TextEdit” in *Inside Macintosh: Text*.
`{anti-alias:}`	Specifies whether text should be displayed using anti-aliasing. Anti-aliasing smooths the edges of the text by blending the edge colors of the text and background. This text descriptor takes one parameter. Possible values are `on` and `off`; the default value is `off`. This corresponds to the `dfAntiAlias` display flag.
`{textColor:}`	Specifies the color of the text. This text descriptor takes three parameters: red, green, and blue color values. Each parameter should be between 0 and 65535. For example, the text descriptor `{textColor:45000,0,0}` sets the text color to a shade of red.
`{backColor:}`	Specifies the background color of the region specified by the `{textBox:}` text descriptor or the region specified by the `{height:}` and `{width:}` descriptors. This text descriptor takes three parameters: red, green, and blue color values. Each parameter should be between 0 and 65535. For example, the text descriptor `{backColor:0,45000,0}` sets the background color to a shade of green.

`{hiliteColor:}`	This display flag specifies the color to be used to highlight text. This text descriptor takes three parameters: red, green, and blue color values. Each parameter should be between 0 and 65535. For example, the text descriptor `{hiliteColor:45000,0,0}` sets the highlight color to a shade of red.
`{inverseHilite:}`	This display flag specifies whether to highlight text using reverse video instead of the highlight color. This text descriptor takes one parameter. Possible values are `on` and `off`; the default value is `off`. This corresponds to the `dfInverseHilite` display flag.
`{keyedText:}`	This display flag specifies whether text should be rendered over the background without drawing the background color. This technique is also known as masked text. This text descriptor takes one parameter. Possible values are `on` and `off`; the default value is `off`. This corresponds to the `dfKeyedText` display flag.
`{hilite:}`	Specifies characters to be highlighted in a text sample. This text descriptor takes two parameters, the first and last characters to highlight in the sample. For example, the text descriptor `{hilite: 11, 14}` highlights the word “text” in the text sample “This is a text track”.
`{textColorHilite:}`	This display flag specifies whether text should be highlighted by changing the color of the text. This text descriptor takes one parameter. Possible values are `on` and `off`; the default value is `off`.
`{karaoke:}`	Highlights groups of characters in the subsequent text sample at specified times. For example, you use this text descriptor to highlight the words in a song while playing the song’s sound track. The parameters for this text descriptor are specified as a set of tuples of the form (*time value*, *starting offset*, *ending offset*), separated by semicolons. The set of tuples is preceded by the total number of tuples. For each tuple, at the specified time value, the text from the starting offset to the ending offset is highlighted.
`{continuousKaraoke:}`	This display flag specifies whether karaoke should ignore the starting offset and highlight all text from the beginning of the sample to the ending offset. Possible values are `on`

	and `off`; the default value is `off`. If continuous karaoke is not enabled, karaoke highlights one offset range at a time. In order for this text descriptor to take effect, the karaoke text descriptor must be in effect. This corresponds to the `dfContinuousKaraoke` display flag.
`{dropShadow:}`	This display flag specifies whether the text sample supports drop shadows. This text descriptor takes one parameter. Possible values are `on` and `off`; the default value is `off`. This corresponds to the `dfDropShadow` display flag.
`{dropShadowOffset:}`	Specifies an offset for the drop shadow. This text descriptor takes two parameters, an offset to the right and an offset down. For example, the text descriptor `{dropShadowOffset: 3, 4}` offsets the drop shadow 3 pixels to the right and 4 pixels down. The default is `{dropShadowOffset: 6, 6}`. In order for this text descriptor to take effect, drop shadowing must be enabled.
`{dropShadowTransparency:}`	Specifies the intensity of the drop shadow. This text descriptor takes one parameter, a value between 0 and 255. The default value is 127. In order for this text descriptor to take effect, drop shadowing must be enabled.

Time stamps

`{timeStamps:}`	Specifies whether time stamps are absolute or relative. This text descriptor takes one parameter. Possible values are `absolute` and `relative`. For example, `{timestamps:absolute}` specifies absolute time stamps. If time stamps are absolute, the time stamp is specified relative to the start of the track for each sample. If time stamps are relative, the time stamp is specified relative to the previous time stamp for each sample.
`{timeScale:}`	Specifies the time scale for the text track. This time scale is used to calculate the mantissa for a time stamp. For example, if the text descriptor `{timeScale:600}` is specified, the time stamp [00:00:07.300] would be interpreted as 7.5 seconds. (Here, you have 300 divided by 600, which equals 0.5). The maximum value for this parameter is 10000.

Scrolling

`{scrollIn:}`	This display flag specifies whether text should be scrolled in until the last of the text is in view. This text descriptor takes one parameter. Possible values are `on` and `off`; the default value is `off`. If both `{scrollIn:}` and `{scrollOut:}` are set, the text is scrolled in and then scrolled out. If the `{scrollDelay:}` text descriptor is set, text is scrolled using the specified delay. This corresponds to the `dfScrollIn` display flag.
`{scrollOut:}`	This display flag specifies whether text should be scrolled out until the last of the text is in view. This text descriptor takes one parameter. Possible values are `on` and `off`; the default value is `off`. If both `{scrollIn:}` and `{scrollOut:}` are set, the text is scrolled in and then scrolled out. If the `{scrollDelay:}` text descriptor is set, text is scrolled using the specified delay. This corresponds to the `dfScrollOut` display flag.
`{horizontalScroll:}`	This display flag specifies whether to scroll a single line of text horizontally. This text descriptor takes one parameter. Possible values are `on` and `off`; the default value is `off`. If you do not specify this descriptor, the scrolling is vertical. The scrolling direction is determined by the `{reverseScroll:}` text descriptor. This corresponds to the `dfHorizScroll` display flag.
`{reverseScroll:}`	This display flag specifies whether to reverse the direction of scrolling. This text descriptor takes one parameter. Possible values are `on` and `off`; the default value is `off`. For vertical scrolling, the default direction is up. For horizontal scrolling, the default direction is left. For example, if you specify `{reverseScroll:on}` and `{horizontalScroll:off}`, the text is scrolled vertically down. This corresponds to the `dfReverseScroll` display flag.
`{continuousScroll:}`	This display flag specifies whether new samples should cause previous samples to scroll out. This text descriptor takes one parameter. Possible values are `on` and `off`; the default value is `off`. In order for this text descriptor to take effect, either `{scrollIn:}` or `{scrollOut:}` must be enabled. This corresponds to the `dfContinuousScroll` display flag.
`{flowHorizontal:}`	This display flag specifies whether text flows within the text box when it is scrolled horizontally. This text descriptor

takes one parameter. Possible values are `on` and `off`; the default value is `off`. For example, if you specify `{flowHorizontal:off}`, the text flows as if the text box had no right edge. In order for this text descriptor to take effect, horizontal scrolling must be enabled. This corresponds to the `dfFlowHoriz` display flag.

`{scrollDelay:}`	This display flag specifies a scroll delay for the sample. This text descriptor takes one parameter, the number of units of the delay in the text track's time scale. For example, if the time scale is 600, the text descriptor `{scrollDelay: 600}` causes subsequent text to be delayed one second. In order for this text descriptor to take effect, scrolling must be enabled.

Time Stamps

When you export text and text descriptors from a text track, the text component also exports a time stamp for each sample. The time stamp indicates the starting and ending time of the sample, either relative to the start of the movie (`kMovieExportAbsoluteTime`) or to the end of the previous sample (`kMovieExportRelativeTime`). On import, the time stamps maintain the timing positions of the text samples relative to other media in the movie.

The format of a time stamp is

```
[HH:MM:SS.xxx]
```

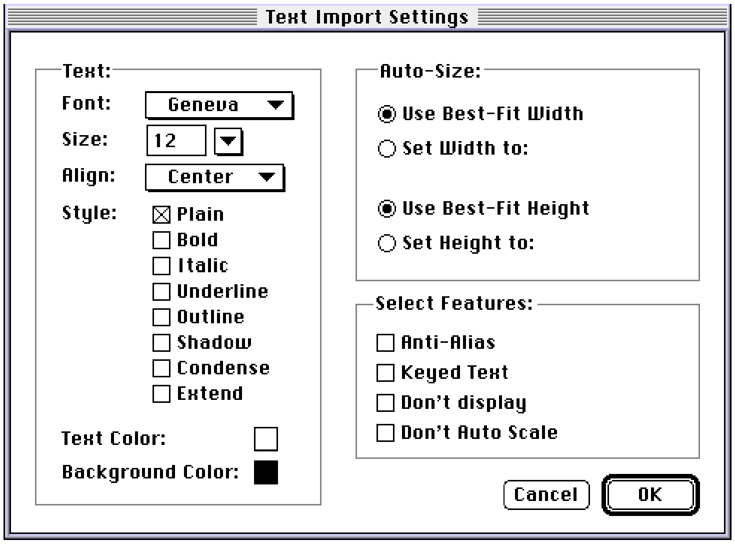
where `HH` represents the number of hours, `MM` represents the number of minutes, `SS` represents the number of seconds, and `xxx` represents the mantissa (the fractional part of a second). The mantissa is expressed in the time scale of the text track. For example, if the time scale of the text track is 600, the time stamp `[00:00:07.300]` is interpreted as 7.5 seconds. If the time scale of the text track is 10, the time stamp `[00:00:07.5]` is also interpreted as 7.5 seconds. The maximum time scale for a text track is 10000.

When a text export component exports a text sample, it first exports the time stamp, followed by a return character. Then, it exports the sample's text and text descriptors. If a text sample does not contain any text, the text component exports the time stamp and return character, but no text.

Importing Text

When you import text, you can override the text descriptors in the text file by specifying options in the Text Import Settings dialog box, shown in Figure 11-2. On import, the settings specified in the dialog box are applied to all imported samples. To display this dialog box programmatically, you can call the `MovieImportDoUserDialog` function, described in *Inside Macintosh: QuickTime Components*.

Figure 11-2 Text import settings dialog box


Text Import Settings

Text:
Font: Geneva
Size: 12
Align: Center
Style: [✓ Plain] Bold Italic Underline Outline Shadow Condense Extend
Text Color: □
Background Color: ■

Auto-Size:
[✓ Use Best-Fit Width] Set Width to:
[✓ Use Best-Fit Height] Set Height to:

Select Features:
Anti-Alias
Keyed Text
Don't display
Don't Auto Scale

Importing In Place

Some movie data import components can create a movie from a file without having to write to a separate disk file. Examples include MPEG, AIFF, DV, and AVI import components; data in files of these types can be played directly by the appropriate media handler components without any data conversion. In such cases it is inappropriate for the user to have to specify a destination file, because there is no need for such a file.

If your import component can operate in this manner, set the `canMovieImportInPlace` flag to 1 in your component flags when you register your component. The standard file dialog box uses this flag to determine how to import files. The `OpenMovieFile` and `NewMovieFromFile` functions use this flag to open some kinds of files as movies.

Audio CD Import Component

QuickTime 1.6.1 introduced the audio CD import component. This movie import component allowed users to open audio CD tracks from the QuickTime standard file preview dialog box, then convert and save the audio as a movie.

In QuickTime 2.1, the audio CD import component was revised to create AIFF files, rather than movie files. These files also contain movie resources, so you can open them as movies.

When you open an audio track on an Apple CD-ROM drive (or equivalent), the Open button changes to a Convert button. When you click Convert, the audio CD import options dialog box appears. Use this dialog box to configure the sound settings. You can specify the sample rate, sample size, and channel settings. You can also select the portion of the track that you want to convert.

DV Video Import and Export Components

QuickTime 3 includes movie data exchange components for DV video. These components are described in the following sections.

DV Movie Import Component

The DV movie import component converts a file containing DV video data into a QuickTime movie. The input file must be a Mac OS file of type `'dvc!'` or a Windows file with the `.dv` file extension. The output file contains a QuickTime movie with two tracks:

- A video track whose samples are of type `kDVNTSCCodecType` for NTSC video data or `kDVCPALCodecType` for PAL video data.
- A sound track whose samples are of type `kDVAudioFormat`.

The data is converted in place, as described in “Importing In Place” (page 351), and the import operation typically takes less than a second. Because both tracks in the QuickTime file refer to the same data, flattening the file creates a file that is twice the size of the original DV data.

You can perform the same operations on the resulting QuickTime movie—including playback, editing, and stepping—that you can for other QuickTime movies. Because video and audio are interleaved in the underlying data, applications for editing movies should make it possible to create a separate file that contains only the audio data for the movie. This can be done by calling the `ConvertMovieToFile` function and specifying `kQTFileTypeAIFF` as the destination file format.

DV Movie Export Component

The DV movie export component converts a file containing a QuickTime movie to a file containing DV data for the movie.

Note
The input file must contain a video track; the DV movie export component cannot convert a movie that contains only audio. ◆

If the video track in the QuickTime movie is already in DV format, the DV movie export component does not recompress the video. This makes it possible to edit DV video in QuickTime and then export it without any loss of video quality due to recompression.

Exporting DV Data from an Application

An application can export DV data without creating a QuickTime movie file by using a callback procedure to supply media data, as described in “Exporting Data from Sources Other Than Movies” (page 353).

Exporting Data from Sources Other Than Movies

A movie data export component can now export data from sources other than QuickTime movies. To do this, the software that exports data must implement callback functions that provide services to the movie data export component. The callback functions and other functions that support this feature are described in this section.

For more information on movie data export components, see Chapter 9, “Movie Data Exchange Components,” in *Inside Macintosh: QuickTime Components*.

Using a Movie Data Export Component to Export Audio

The following example illustrates how to use a movie data export component to export audio data to an AIFF file.

Instantiating the Data Export Component

The first step in using a movie data export component to create an AIFF file is instantiating an AIFF data export component. An example of this is shown in Listing 11-2.

Listing 11-2 Instantiating a data export component

```
ComponentDescription cd;
MovieExportComponent ci;
cd.componentType = MovieExportType;
cd.componentSubType = 'AIFF';
cd.componentManufacturer = 0;
cd.componentFlags = canMovieExportFromProcedures;
cd.componentFlagsMask = canMovieExportFromProcedures;
ci = OpenComponent(FindNextComponent(nil, &cd));
```

Configuring the Data Export Component

Once an AIFF movie data export component has been instantiated, it must be configured to open a single output audio stream. Listing 11-3 is an example of creating an output audio stream by calling `MovieExportAddDataSource`. In this example, `MovieExportAddDataSource` also provides the callback functions for supplying media data.

Listing 11-3 Configuring the audio export component

```
#define kMySampleRate 22050
#define kSoundBufferSize 1024
typedef struct
{
    Ptr soundData;
    SoundDescriptionHandle soundDescription;
    long trackID;
}
MyReferenceRecord;
MyReferenceRecord myRef;
SoundDescriptionPtr sdp;
myRef.soundData = NewPtr(kSoundBufferSize);
myRef.soundDescription = NewHandleClear(sizeof(SoundDescription));
sdp = *myRef.soundDescription;
sdp->descSize = sizeof(SoundDescription);
sdp->dataFormat = kRawCodecType;
sdp->numChannels = 1;
sdp->sampleSize = 8;
sdp->sampleRate = kMySampleRate << 16;
MovieExportAddDataSource(ci, SoundMediaType, kMySampleRate,
        &myRef.trackID, getSoundTrackPropertyProc,
        getSoundTrackDataProc, &myRef);
```

Note
On Macintosh PowerPC computers, the `getSoundTrackPropertyProc` and `getSoundTrackDataProc` routines should be universal procedure pointers (UPPs). ◆

Exporting the Data

The export operation takes place when all of the required output tracks have been created.

Listing 11-4 Exporting audio data

```
StandardFileReply reply;
Handle dataRef;
// get output file from user
QTNewAlias(&reply.sfFile, (AliasHandle *)&dataRef, true);
MovieExportFromProceduresToDataRef(ci, dataRef, rAliasType);
```

`MovieExportFromProceduresToDataRef` calls the two functions specified in `MovieExportAddDataSource` to obtain data to generate the output file. The first function returns information about the output track's properties, including the sample rate and supported media. If no value is returned for a particular property, the exporter specifies a default value based on the source data format.

In the example in Listing 11-5, the output sample rate is set at 32000 Hz, with all other properties left unspecified.

Listing 11-5 Obtaining output track information

```
pascal OSErr getSoundTrackPropertyProc(void *refcon, long trackID,
    OSType propertyType, void *propertyValue)
{
    OSErr err = noErr;
    switch (propertyType)
    {
        case scSoundSampleRateType:
            *(Fixed *)propertyValue = 32000L << 16;
            break;
        default:
            err = paramErr;
            break;
    }
    return err;
}
```

The second function provides data to be exported. Listing 11-6 shows a block of sound data (silence) returned for each export request. The export operation ends when this function returns `eofErr`.

Listing 11-6 Providing output track information to the export component

```
pascal OSErr getSoundTrack(void *refCon,
        MovieExportGetDataParams *params)
{
    MyReferenceRecord *myRef = (MyReferenceRecord *)refCon;
    if (params->requestedTime > kMySampleRate * 10)
        return eofErr;                  // end of data after 10 seconds
    params->dataPtr = myRef->soundData;
    params->dataSize = kSoundBufferSize;
    params->actualTime = params->requestedTime;
    params->sampleCount = kSoundBufferSize;
    params->durationPerSample = 1;
    params->descType = SoundMediaType;
    params->descSeed = 1;
    params->desc = (SampleDescriptionHandle)myRef->soundDescription;
    return noErr;
}
```

Using a Movie Data Export Component to Export Video

Using a movie data export component to create a QuickTime movie file is similar in many respects to creating an AIFF file, as shown in the previous example. Media data is handled differently in each case, however.

Instantiating the Data Export Component

Listing 11-7 illustrates the first step, instantiating a movie data export component for video data.

Listing 11-7 Instantiating a movie data export component

```
ComponentDescription cd;
MovieExportComponent ci;
cd.componentType = MovieExportType;
cd.componentSubType = ‘MooV’;
cd.componentManufacturer = ‘appl’;
cd.componentFlags = canMovieExportFromProcedures;
cd.componentFlagsMask = canMovieExportFromProcedures;
ci = OpenComponent(FindNextComponent(nil, &cd));
```

Configuring the Data Export Component

Listing 11-8 illustrates the next step: configuring the data export component to create a single output video track. The call to `MovieExportAddDataSource` provides the callback functions for supplying media data.

Listing 11-8 Configuring the movie data export component

```
#define kMySampleRate 2997                  /* 29.97 fps */
#define kMyFrameDuration 100                /* one frame at 29.97 fps */
typedef struct
{
    GWorldPtr gw;
    ImageDescriptionHandle imageDescription;
    long trackID;
}
MyReferenceRecord;
MyReferenceRecord myRef;
myRef.gw = nil;
myRef.imageDescription = nil;
MovieExportAddDataSource(ci, VideoMediaType, kMySampleRate,
        &myRef.trackID, getVideoPropertyProc,
        getVideoDataProc, &myRef);
```

Exporting the Data

At this point, operations could be added. For example, additional video tracks or sound tracks could be created. Once all the output tracks are created, the export operation takes place. This is the same as in the AIFF export example.

Listing 11-9 Exporting video data

```
StandardFileReply reply;
Handle dataRef;
// get output file from user
QTNewAlias(&reply.sfFile, (AliasHandle *)&dataRef, true);
MovieExportFromProceduresToDataRef(ci, dataRef, rAliasType);
```

The `getVideoPropertyProc` function returns information about the output track's properties (for example, the compression format). If the function doesn't return a value for a particular property, the exporter will choose a default value based (usually) on the source data format. In Listing 11-10, dimensions are set to 160x120 and the compression format is set to JPEG. All other properties are unspecified.

Listing 11-10 Obtaining information about output track properties

```
pascal OSErr getVideoPropertyProc(void *refcon, long trackID,
    OSType propertyType, void *propertyValue)
{
    OSErr err = noErr;
    switch (propertyType) {
        case meWidth:
            *(Fixed *)propertyValue = 160L << 16;
            break;
        case meHeight:
            *(Fixed *)propertyValue = 120L << 16;
            break;
        case scSpatialSettingsType:
            {
            SCSpatialSettings *ss = propertyValue;
            ss->codecType = kJPEGCodecType;
            ss->codec = 0;
            ss->depth = 0;
            ss->spatialQuality = codecNormalQuality;
            }
            break;
        default:
            err = paramErr;
            break;
    }
    return err;
}
```

The `videoGetDataProc` function provides video frames to the export operation. In the example in Listing 11-11, the same blank frame is returned for each request. The export operation ends when this function returns `eofErr`. Any data allocated by `videoGetDataProc` must be disposed of after the export operation is complete.

Listing 11-11 Providing video frames for export

```
pascal OSErr videoGetDataProc(void *refCon,
        MovieExportGetDataParams *params)
{
    OSErr err = noErr;
    MyReferenceRecord *myRef = (MyReferenceRecord *)refCon;
    TimeRecord tr;
    if (params->requestedTime > kMySampleRate * 10)
        return eofErr;// end of data after 10 seconds
    if (!myRef->gw) {
        Rect r;
        CGrafPtr savePort;
        GDHandle saveGD;
        SetRect(&r, 0, 0, 320, 240);
        NewGWorld(&myRef->gw, 32, &r, nil, nil, 0);
        LockPixels(myRef ->gw->portPixMap);
        MakeImageDescriptionForGWorld(myRef->gw,
                &myRef->imageDescription);
        GetGWorld(&savePort, &saveGD);
        SetGWorld(myRef->gw, nil);
        EraseRect(&r);
        SetGWorld(savePort, saveGD);
    }
    params->dataPtr = GetPixBaseAddr(myRef->gw->portPixMap);
    params->dataSize = (**myRef->imageDescription).dataSize;
    params->actualTime = params->requestedTime;
    params->descType = VideoMediaType;
    params->descSeed = 1;
    params->desc = (SampleDescriptionHandle)
                myRef->imageDescription;
    params->durationPerSample = kMyFrameDuration;
    params->sampleFlags = 0;
}
```

Implementing Movie Data Exchange Components

The following section discusses how you can implement new component routines to save and restore component settings available in QuickTime 3.

Save-and-Restore Component Routines

If you are writing movie data exchange components, and would like your components' settings to be saved and restored, you need to implement two additional component routines in order to allow your components to have their settings saved and restored. A component's settings are stored in a QuickTime QT atom container. The data stored in the QT atom container is private to the particular component but should be stored so that it is possible to read the data on all platforms supported by QuickTime, thus allowing the same settings to be used anywhere. ("Settings Container Format and Guidelines" (page 362) provides further details.) For each type of movie data exchange component, there is one routine to return a QT atom container holding the settings and another routine to configure the component from previously-saved settings. For more information about QT atom containers, see "QuickTime Atoms" (page 47).

Import component developers need to implement the `MovieImportGetSettingsAsAtomContainer` and `MovieImportSetSettingsFromAtomContainer` routines. For `MovieImportGetSettingsAsAtomContainer`, the component should allocate a new QT atom container, stuff current settings into it, and return it to the caller. For `MovieImportSetSettingsFromAtomContainer`, the component should accept a QT atom container, extract the settings in which it is interested, and change its internal state.

Export component developers need to implement the `MovieExportGetSettingsAsAtomContainer` and `MovieExportSetSettingsFromAtomContainer` routines. Like import components, the component's `MovieExportGetSettingsAsAtomContainer` routine allocates and returns a QT atom container holding the component's settings. For `MovieExportSetSettingsFromAtomContainer`, the component accepts a QT atom container, extracts the settings, and updates its internal state.

Settings Container Format and Guidelines

The particular atoms stored within the component's settings QT atom container are private to that component type. However, there are some guidelines that need to be followed. These include:

- In all `SetSettingsFromAtomContainer` routines, the QT atom container belongs to the caller. The component should not dispose of the passed QT atom container.
- The settings QT atom container should contain one or more top-level atoms. These top-level atoms can contain either leaf data or other atoms. Each atom has both a type (`QTAtomType`) and an ID. Choosing an atom type that is mnemonic is helpful in indicating how it is used. For example, QuickTime stores video compression settings in atoms of type `'vide'`. Sound compression settings are stored in `'soun'` atoms. The text components use `'text'` for their atom types.
- Several of QuickTime's export components use the standard compression component to allow the user to configure compression settings for exported files. When one of these components is asked to return its settings atom container, the export component first requests that the standard compression component return its settings using the `SCGetSettingsAsAtomContainer` function described above. To the QT atom container it receives, the export component adds any of its own settings. When the export component's `SetSettingsFromAtomContainer` is called, the exporter calls `SCSetSettingsFromAtomContainer` with the passed atom container. The standard compression component extracts only those settings it expects, ignoring all other, and configures itself. The exporter then looks for its own settings in the same atom container and configures itself.

  This is possible because the standard compression and data exchange components both use QT atom containers to hold their settings. Because many third-party developers do the same, there must be a mechanism so that QuickTime's own top-level atom types and those of third parties don't collide. To achieve this, Apple Computer reserves all top-level atom types consisting exclusively of lowercase letters with or without numerals. For example, `'vide'` is reserved by Apple, but `'Vide'` is not. There is no restriction on the atom types for atoms stored within these top-level atoms.
- Apple recommends that you store all of your component settings under a single top-level atom. However, there is no requirement to do so.

- The data within an atom should be stored in a canonical form on all platforms. It should be always in big-endian format or always in little-endian format. Different types of atoms can be stored in different endian orders but for a single type of atom, it should always use the same order. This allows the settings to be created in the Mac OS and read in Windows or vice-versa.
- In either `MovieImportSetSettingsFromAtomContainer` or `MovieExportSetSettingsFromAtomContainer`, you should not necessarily expect all atoms to be included in the atom container you receive. This allows another developer, for example, to create a settings atom container, add atoms and data for only those parts of the settings that should be changed, and then pass this incomplete atom container to the component. The component then only changes those particular settings, leaving other settings alone. QuickTime's own components use this approach.
- If `nil` is passed for the settings to the component routines, return `paramErr`.
- If your component does not have configurable settings, you do not need to implement the settings-related routines.

Implementing Movie Data Import Components

Specifying a Part of a File to Import

When using certain movie import components, applications can import data from a part of a file rather than the entire file by calling the `MovieImportSetOffsetAndLimit` function. This function allows an application to specify a byte offset into the file at which the import operation begins and another offset, known as the *limit,* that indicates the last data in the file that can be imported. This function is especially useful when one file format is embedded in another; it allows an application to skip header data for the enclosing file and begin importing data at the start of the desired format.

Not all movie import components support the `MovieImportSetOffsetAndLimit` function. Those that do include the movie import components provided with QuickTime for the `kQTFileTypeAIFF`, `kQTFileTypeWave`, and `kQTFileTypeMuLaw` file types. Those that do not return the result code `badComponentSelector` in response to a `MovieImportSetOffsetAndLimit` call.

Getting a List of Supported MIME Types

Applications can get a list of MIME types supported by a movie import component by calling the `MovieImportGetMIMETypeList` function.

Registering Movie Data Export Components

In previous versions of QuickTime, it was not possible to have more than one export (`'spit'`) component registered for the same type of file and the same track media type. Additionally, it is not possible to have more than one component that exports to a type of file that considers from the whole movie. What this meant was that if two separate exporters wanted to export to the `'PICT'` file format, for example, only one could be registered at a time.

In the past, the distinction between a media type-specific (or track type-specific) exporter and a movie-wide exporter was based on the `componentManufacturer` value with which the component was registered. Media type-specific components stored the type of the media handler (`'mhlr'`) component in this field. Movie-wide components used 0 in this field.

QuickTime 3 now allows more than one export (`'spit'`) component to be registered for the same type of file and same export source (the movie or the particular track type). This is accomplished in a way that preserves compatibility with third-party components that may have already been written using the former rules.

Introducing a New Registration Mechanism

QuickTime 3 introduces a new movie export component routine that returns the same information that would have been previously stored in the `componentManufacturer` field of the registered `'spit'` components. A new export-specific component flag indicates that the export component implements the new protocol. This enables developers and QuickTime to differentiate between older components and those using the newer mechanism. By implementing the routine, the export component's `componentManufacturer` field can be used to differentiate components.

The new routine is `MovieExportGetSourceMediaType`. This routine returns an `OSType` value through its `mediaType` parameter, which is interpreted in exactly the same way that the `componentManufacturer` was previously interpreted. If the export component requires a particular type of track to exist in a movie, it returns that media handler type (e.g., `VideoMediaType`, `SoundMediaType`, etc.)

through the `mediaType` argument. If the export component works for an entire movie, it returns 0 through this parameter.

```
EXTERN_API( ComponentResult )
    MovieExportGetSourceMediaType   (MovieExportComponent   ci,
                                 OSType * mediaType);
```

The following component flag indicates that this routine is implemented:

```
movieExportMustGetSourceMediaType = 1L << 19,
```

If you implement the `MovieExportGetSourceMediaType` routine, you must register the component with this flag. Otherwise, the Movie Toolbox will not know to call the routine and will assume the older semantics for the `componentManufacturer` field.

As in the past, using this mechanism does not replace the need for implementing `Validate` in your export components. The new mechanism is only used to find candidate components.

Changes to QuickTime 3 Export Components

Both the Movie and the DVC export components use the new export registration mechanism. The components are now registered as shown below.

QuickTime Movie

```
componentType                'spit'
componentSubType             'MooV'
componentManufacturer        'appl'
componentFlags               canMovieExportFiles
                                + canMovieExportFromProcedures
                                + hasMovieExportUserInterface
                                + canMovieExportValidateMovie
                                + movieExportMustGetSourceMediaType
```

DV Stream

```
componentType                'spit'
componentSubType             'dvc!'
componentManufacturer        'appl'
componentFlags               canMovieExportFiles
```

```
                                    + canMovieExportFromProcedures
                                    + hasMovieExportUserInterface
                                    + canMovieExportValidateMovie
                                    + movieExportMustGetSourceMediaType
```

Because the DVC component uses the QuickTime Movie export component, it has been changed now to search for the `'MooV'` exporter, using the following `ComponentDescription` values:

```
    cd.componentType = 'spit';
    cd.componentSubType = MovieFileType;
    cd.componentManufacturer = 'appl';
    cd.componentFlags = canMovieExportFromProcedures +
                        movieExportMustGetSourceMediaType;
    cd.componentFlagsMask = cd.componentFlags;
```

If you are working with export components—either writing them, or trying to enumerate or otherwise match up components with source media types—you need to take into account these changes in QuickTime 3. If you are writing export components, you should move to this new registration mechanism. If you are enumerating components, you need to make appropriate changes. If you are not doing either, you won't need to be concerned about the changes because the QuickTime Movie Toolbox has been updated and will hide the details.

Implementing Movie Data Export Components

You can now implement a movie data export component by calling the new `MovieExportFromProceduresToDataRef` function.

Because many existing applications expect to be able to perform an export operation from a movie or track, export components should support `MovieExportToFile`, `MovieExportFromProceduresToDataRef` and `MovieExportToDataRef`. Using the routines described in "Functions Provided by the Movie Export Component" (page 392), it is possible to provide small implementations of these old-style routines that simply call the newer `MovieExportFromProceduresToDataRef` to perform the actual operation.

Listing 11-12 shows how to implement `MovieExportToFile` so that it simply calls `MovieExportToDataRef`.

Listing 11-12 Calling `MovieExportToDataRef` from `MovieExportToFile`

```
pascal ComponentResult MovieExportToFile(Globals store,
    const FSSpec *theFile, Movie m, Track onlyThisTrack,
    TimeValue startTime, TimeValue duration)
{
    ComponentResult err;
    AliasHandle alias;
    err = QTNewAlias (theFile, &alias, true);
    err = MovieExportToDataRef(store->self, (Handle)alias,
        rAliasType, m, onlyThisTrack, startTime, duration);
    DisposeHandle((Handle)alias);
}
```

Listing 11-13 example shows how to use the utility routines provided by the QuickTime movie data export component to implement `MovieExportToDataRef` by calling `MovieExportFromProceduresToDataRef`. Your implementation may differ, depending on the types of data to be exported. For example, the number and type of data sources created may change. This example creates a single sound data source, and is appropriate for any movie data export component that exports audio only.

Listing 11-13 Calling `MovieExportFromProceduresToDataRef` from `MovieExportToDataRef`

```
pascal ComponentResult MovieExportToDataRef(Globals store,
    Handle dataRef, OSType dataRefType, Movie m,
    Track onlyThisTrack, TimeValue startTime,
    TimeValue duration)
{
    ComponentResult err;
    ComponentDescription cd;
    ComponentInstance ci;
    TimeScale scale;
    MovieExportGetPropertyUPP getPropertyProc = nil;
    MovieExportGetDataUPP getDataProc = nil;
    void *refCon;
    long trackID;
    cd.componentType = MovieExportType;
    cd.componentSubType = 'MooV';
    cd.componentManufacturer = 0;
    cd.componentFlags = canMovieExportFromProcedures;
    cd.componentFlagsMask = canMovieExportFromProcedures;
    err = OpenAComponent(FindNextComponent(nil, &cd), &ci);
    err = MovieExportNewGetDataAndPropertiesProcs(ci,
        SoundMediaType, &scale, m, onlyThisTrack,
        startTime, duration,
        &getPropertyProc, &getDataProc, &refCon);
    err = MovieExportAddDataSource(store->self, SoundMediaType,
            scale, &trackID, getPropertyProc, getDataProc,
            refCon);
    err = MovieExportFromProceduresToDataRef(store->self,
            dataRef, dataRefType);
}
```

Note
The code in Listing 11-13 retrieves the default property and data procedures instead of providing them. It also must dispose of them. ◆

Determining Whether Movie Data Export Is Possible

Although a movie export component can export one or more media types, it may not be able to export all the kinds of data stored in those media. Applications can find out whether a movie export component can export all the data for a particular movie or track by calling the `MovieExportValidate` function.

Not all export components implement the `MovieExportValidate` call. In the following code snippet, you make the `Validate` call, and even if it is not implemented it is still true.

```
Boolean canExport = true;
    MovieExportValidate(ci, &canExport);
    if(canExport) {
    . . .
    }
```

Movie Data Exchange Components Reference

This section describes new constants, data types, and functions of movie data exchange components.

Constants

Flags for Movie Import and Export Components

QuickTime 1.6.1 added four new `componentFlags` values. The `canMovieImportInPlace` and `movieImportSubTypeIsFileExtension` values were added in QuickTime 2.0:

```
enum {
    canMovieExportAuxDataHandle             = 1 << 7,
    canMovieImportValidateHandles           = 1 << 8,
    canMovieImportValidateFile              = 1 << 9,
    dontRegisterWithEasyOpen                = 1 << 10,
    canMovieImportInPlace                   = 1 << 11,
    movieImportSubTypeIsFileExtension       = 1 << 12,
    canMovieImportPartial                   = 1 << 13,
    hasMovieImportMIMEList                  = 1 << 14,
    canMovieExportFromProcedures            = 1 << 15,
    canMovieExportValidateMovie             = 1L << 16,
    movieExportNeedsResourceFork            = 1L << 17,
    canMovieImportDataReferences        = 1L << 18,
    movieExportMustGetSourceMediaType   = 1L << 19
};
```

Constant descriptions

`canMovieExportAuxDataHandle`
: Set this bit if your export component exports to an auxiliary data handle. A movie export component that supports the `MovieExportGetAuxillaryData` function should set this flag in its `componentFlags` parameter.

`canMovieImportValidateHandles`
: Set this bit if your movie import component can and wants to validate handles.

`canMovieImportValidateFile`
: Set this bit if your movie import component can and wants to validate files.

`dontRegisterWithEasyOpen`
: Set this bit when Macintosh Easy Open is installed and you do not want QuickTime to register your component with Easy Open (preferring to handle interactions with Macintosh Easy Open yourself).

`canMovieImportInPlace`
: Set this bit if your movie import component can create a movie from a file without having to write to a separate disk file. Examples include MPEG and AIFF import components.

`movieImportSubTypeIsFileExtension`
: Set this bit if your component's subtype is a file extension instead of a Macintosh file type. For example, if your import component opens files with an extension of `.doc`, you would set this flag and set your component subtype to `'DOC '`.

`canMovieImportPartial`
: Set this bit if your movie import component implements the `MovieImportSetOffsetAndLimit` function.

`hasMovieImportMIMEList`
: Set this bit if you movie import component supports returning a MIME type list. A movie import component that supports the `MovieImportGetMIMETypeList` function should set the `hasMovieImportMIMEList` flag in the `componentFlags` field of the component description record.

`canMovieExportFromProcedures`
: Set this bit if your movie export component supports the `MovieExportFromProceduresToDataRef` function. A movie export component that supports the `MovieExportFromProceduresToDataRef` function should also set the `canMovieExportFromProcedures` flag in its `componentFlags`.

`canMovieExportValidateMovie`

Set this bit if your movie export component validates the movie passed to it. A movie export component that supports the `MovieExportValidate` function should also set the `canMovieExportValidateMovieflag` in its component flags.

`movieExportNeedsResourceFork`

Set this bit if your export component requires the creation of the resource fork of a Mac OS file.

`canMovieImportDataReferences`

Set this bit if your movie import component can accept a data reference (such as a handle) as the source for the import.

`movieExportMustGetSourceMediaType`

Set this bit if the export component implements the new exporter component registration mechanism. The component must implement the `MovieExportGetSourceMediaType` function.

Text Export Options

The following constants represent options for exporting text using a text export component. You use these constants to specify the format of the exported text. From the QuickTime user interface, you specify a text export option in the Text Export Settings dialog box. You can also specify the text export option programmatically by calling `TextExportSetSettings` (page 378).

```
enum {
    kMovieExportTextOnly        = 0,
    kMovieExportAbsoluteTime    = 1,
    kMovieExportRelativeTime    = 2
};
```

Constant descriptions

`kMovieExportTextOnly`

Export text only, without text descriptors or time stamps.

`kMovieExportAbsoluteTime`

Export text with text descriptors and time stamps. For each sample, calculate the time stamp relative to the start of the movie.

`kMovieExportRelativeTime`
: Export text with text descriptors and time stamps. For each sample, calculate the time stamp relative to the previous time stamp.

Data Types

MIME Type List

The `MovieImportGetMIMETypeList` function returns a list of the MIME types supported by a movie import component. This list is contained in the QT atom container described in this section. For more information about QT atom containers, see "QuickTime Atoms" (page 47).

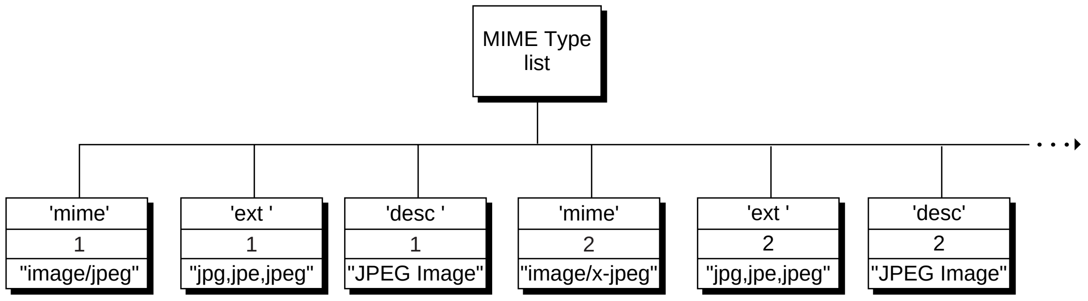
At the top level of the atom container are three atoms for each supported MIME type. The atoms whose IDs are 1 describe the first supported MIME type, the atoms whose IDs are 2 describe the second supported MIME type, and so on.

- An atom of type `kMimeInfoMimeTypeTag` contains a string that identifies the MIME type, such as `image/jpeg` or `image/x-jpeg`.
- The atom of type `kMimeInfoFileExtensionTag` contains a string that specifies likely file extensions for files of this MIME type, such as `jpg`, `jpe`, and `jpeg`. If there is more than one extension, the extensions are separated by commas.
- The atom of type `kMimeInfoDescriptionTag` contains a string describing the MIME type for end users, such as `JPEG Image`.

Note
These atom types contain neither a Pascal nor a C string.
The atom types are simply ASCII characters; an atom's size
is the number of characters. ◆

Figure 11-3 illustrates a MIME type list.

Figure 11-3 A MIME type list



Text Display Data Structure

The `TextDisplayData` structure contains formatting information for a text sample. When the text export component exports a text sample, it uses the information in this structure to generate the appropriate text descriptors for the sample. Likewise, when the text import component imports a text sample, it sets the appropriate fields in the text display data structure based on the sample's text descriptors.

```
struct TextDisplayData {
    long                    displayFlags;
    long                    textJustification;
    RGBColor                bgColor;
    Rect                    textBox;
    short                   beginHilite;
    short                   endHilite;
    RGBColor                hiliteColor;
    Boolean                 doHiliteColor;
    SInt8                   filler;
    TimeValue               scrollDelayDur;
    Point                   dropShadowOffset;
    short                   dropShadowTransparency;
};
typedef struct TextDisplayData TextDisplayData;
```

Field descriptions

`displayFlags`	Contains flags that represent the values of the following text descriptors: `doNotDisplay`, `doNotAutoScale`,

	`clipToTextBox`, `useMovieBackColor`, `shrinkTextBox`, `scrollIn`, `scrollOut`, `horizontalScroll`, `reverseScroll`, `continuousScroll`, `flowHorizontal`, `dropShadow`, `anti-alias`, `keyedText`, `inverseHilite`, `continuousKaraoke`, and `textColorHilite`. For more information on the text sample display flags, see Chapter 1, “Movie Toolbox,” in this reference and the description of the `AddTextSample` function in *Inside Macintosh: QuickTime.*
`textJustification`	Specifies the alignment of the text in the text box. Possible values are `teFlushDefault`, `teCenter`, `teFlushRight`, and `teFlushLeft`. For more information on text alignment and the text justification constants, see the “TextEdit” chapter of *Inside Macintosh: Text.*
`bgColor`	Specifies the background color of the rectangle specified by the `textBox` field. The background color is specified as an RGB color value.
`textBox`	Specifies the rectangle of the text box.
`beginHilite`	Specifies the one-based index of the first character in the sample to highlight.
`endHilite`	Specifies the one-based index of the last character in the sample to highlight.
`doHiliteColor`	Specifies whether to use the color specified by the `hiliteColor` field for highlighting. If the value of this field is `true`, the highlight color is used for highlighting. If the value of this field is `false`, reverse video is used for highlighting.
`filler`	Reserved.
`scrollDelayDur`	Specifies a scroll delay. The scroll delay is specified as the number of units of delay in the text track’s time scale. For example, if the time scale is 600, a scroll delay of 600 causes the sample text to be delayed one second. In order for this field to take effect, scrolling must be enabled.
`dropShadowOffset`	Specifies an offset for the drop shadow. For example, if the point specified is (3,4), the drop shadow is offset 3 pixels to the right and 4 pixels down. In order for this field to take effect, drop shadowing must be enabled.
`dropShadowTransparency`	Specifies the intensity of the drop shadow as a value

between 0 and 255. In order for this field to take effect, drop shadowing must be enabled.

Movie Data Exchange Component Functions

Exporting Text

This section describes new functions provided by text export components.

TextExportGetDisplayData

The `TextExportGetDisplayData` function retrieves text display information for the current sample in the specified text export component.

```
pascal ComponentResult TextExportGetDisplayData (
                        TextExportComponent ci,
                        TextDisplayData *textDisplay);
```

`ci`	Specifies the text export component for this operation. Applications obtain this reference from the Component Manager's `OpenComponent` function.
`textDisplay`	Contains a pointer to a text display data structure. On return, this structure contains the display settings of the current text sample. For more information, see "Text Display Data Structure" (page 373).

DISCUSSION

You call this function to retrieve the text display data structure for a text sample. The text display data structure contains the formatting information for the text sample. When the text export component exports a text sample, it uses the information in this structure to generate the appropriate text descriptors for the sample. Likewise, when the text import component imports a text sample, it sets the appropriate fields in the text display data structure based on the sample's text descriptors.

RESULT CODES

Component Manager errors, as documented in *Mac OS For QuickTime Programmers*.

TextExportGetTimeFraction

The `TextExportGetTimeFraction` function retrieves the time scale the specified text export component uses to calculate time stamps.

```
pascal ComponentResult TextExportGetTimeFraction (
                     TextExportComponent ci,
                     long *movieTimeFraction);
```

`ci` Specifies the text export component for this operation.

`movieTimeFraction`
Contains a pointer to a 32-bit integer. On return, this integer contains the time scale used in the fractional part of time stamps.

DISCUSSION

You call this function to retrieve the time scale used by the text export component to calculate the fractional part of time stamps. You set a text component's time scale by specifying it in the Text Export Settings dialog box or by calling the `TextExportSetTimeFraction` function.

RESULT CODES

Component Manager errors, as documented in *Mac OS For QuickTime Programmers*.

TextExportSetTimeFraction

The `TextExportSetTimeFraction` function set the time scale the specified text export component uses to calculate time stamps.

```
pascal ComponentResult TextExportSetTimeFraction (
                    TextExportComponent ci,
                    long movieTimeFraction);
```

`ci`	Specifies the text export component for this operation.
`movieTimeFraction`	Specifies the time scale used in the fractional part of time stamps. The value should be between 1 and 10000, inclusive.

DISCUSSION

You call this function to set the time scale used by the text export component to calculate the fractional part of time stamps. You can also set a text component's time scale by specifying it in the text export settings dialog box. You can retrieve a text component's time scale by calling the `TextExportGetTimeFraction` function.

RESULT CODES

Component Manager errors, as documented in *Mac OS For QuickTime Programmers*.

TextExportGetSettings

The `TextExportGetSettings` function retrieves the value of the text export option for the specified text export component.

```
pascal ComponentResult TextExportGetSettings (
                    TextExportComponent ci,
                    long *setting);
```

`ci`	Specifies the text export component for this operation.

`setting` Contains a pointer to a 32-bit integer. On return, this integer contains a constant that represents the current value of the text export option. Possible values are `kMovieExportTextOnly`, `kMovieExportAbsoluteTime`, and `kMovieExportRelativeTime`. For more information, see "Text Export Options" (page 371).

DISCUSSION

You call this function when exporting text to retrieve the current value of the text export option for the specified text export component. If the retrieved text export option is `kMovieExportTextOnly`, the text export component exports text without time descriptors or time stamps. If the retrieved text export option is either `kMovieExportAbsoluteTime` or `kMovieExportRelativeTime`, the text export component exports text along with its text descriptors and time stamps.

RESULT CODES

Component Manager errors, as documented in *Mac OS For QuickTime Programmers*.

TextExportSetSettings

The `TextExportSetSettings` function sets the value of the text export option for the specified text export component.

```
pascal ComponentResult TextExportSetSettings (
                    TextExportComponent ci,
                    long setting);
```

`ci` Specifies the text export component for this operation.

`setting` Specifies a constant that represents the current value of the text export option. Possible values are `kMovieExportTextOnly`, `kMovieExportAbsoluteTime`, and `kMovieExportRelativeTime`. For more information, see "Text Export Options" (page 371).

DISCUSSION

You call this function when exporting text to set the value of the text export option for the specified text export component. To export text only, without time descriptors or time stamps, you should set the `setting` parameter to `kMovieExportTextOnly`. To export text with text descriptors and absolute time stamps, you should set the `setting` parameter to `kMovieExportAbsoluteTime`. To export text with text descriptors and relative time stamps, you should set the `setting` parameter to `kMovieExportRelativeTime`.

RESULT CODES

Component Manager errors, as documented in *Mac OS For QuickTime Programmers*.

Importing Movie Data

This section describes new functions provided by movie data import components.

MovieImportGetFileType

The `MovieImportGetFileType` allows your movie data import component to tell the Movie Toolbox the appropriate file type for the most-recently imported movie file.

```
pascal ComponentResult MovieImportGetFileType (MovieImportComponent ci,
                          OSType *fileType);
```

`ci`	Identifies the toolbox's connection to your movie data import component.
`fileType`	Contains a pointer to an `OSType` field. Your component should place the file type value that best identifies the movie data just imported. For example, Apple's Audio CD movie data import component sets this field to `'AIFF'` whenever it creates an AIFF file instead of a movie file.

DISCUSSION

You should implement this function only if your movie data import component creates files other than QuickTime movies. By default, the toolbox makes new files movies, unless you override that default by providing this function.

Result CODES

`badComponentSelector`	0x80008002	Function not supported

MovieImportGetAuxiliaryDataType

The `MovieImportGetAuxiliaryDataType` function returns the type of the auxiliary data that it can accept. For example, calling the text import component with `MovieImportGetAuxiliaryDataType` indicates that the text import component will use `'styl'` information in addition to `'TEXT'` data.

```
pascal ComponentResult MovieImportGetAuxiliaryDataType(
                    MovieImportComponent ci,
                    OSType *auxType);
```

`ci`	Specifies the movie import component for the request. Applications obtain this reference from the Component Manager's `OpenComponent` function.
`auxType`	A pointer to the type of auxiliary data it can import.

RESULT Codes

`badComponentInstance`	0x80008001	Invalid movie import component instance
`badComponentSelector`	0x80008002	Function not supported

MovieImportGetMIMETypeList

Returns a list of MIME types supported by the movie import component.

```
pascal ComponentResult MovieImportGetMIMETypeList (
                        MovieImportComponent ci,
                        QTAtomContainer *mimeInfo);
```

`ci`	Specifies an instance of a movie import component.
`mimeInfo`	A pointer to a MIME type list, a QT atom container that contains a list of MIME types supported by the movie import component. The MIME type list structure is described in "MIME Type List" (page 372). The caller should dispose of the atom container when finished with it.

DISCUSSION

Your movie import component can support MIME types that correspond to formats it supports. To make a list of these MIME types available to applications or other software, it must implement the `MovieImportGetMIMETypeList` function.

To indicate that your movie import component supports this function, set the `hasMovieImportMIMEList` flag in the `componentFlags` field of the component description record.

RESULT Codes

`noErr`	0	No error

`paramErr`	–50	Invalid parameter specified
`memFullErr`	–108	Not enough memory available
`badComponentSelector`	0x80008002	Component does not support the specified request code

MovieImportValidate

The `MovieImportValidate` function allows your movie data import component to validate the data to be passed to your component.

```
pascal ComponentResult MovieImportValidate(
                        MovieImportComponent ci,
                        const FSSpec *theFile,
                        Handle theData,
                        Boolean *valid);
```

`ci`	Specifies the movie import component for the request. Applications obtain this reference from the Component Manager's `OpenComponent` function.
`theFile`	Specifies the file to validate if the importer imports from files.
`theData`	Specifies the data to validate if the importer imports from handles.
`valid`	Contains a pointer to a Boolean value. If the data or file can be imported, the value returned is `true`. Otherwise, it returns `false`.

DISCUSSION

Movie import components can implement this function to allow applications to determine if a given file or handle to data is acceptable for a particular import component. As this function may be called on many files, the validation process should be as fast as possible.

RESULT Codes

`badComponentInstance`	0x80008001	Invalid movie import component instance
`badComponentSelector`	0x80008002	Function not supported

MovieImportSetOffsetAndLimit

The `MovieImportSetOffsetAndLimit` function allows your application to import data from a part of a file rather than the entire file.

```
pascal ComponentResult MovieImportSetOffsetAndLimit(
                        MovieImportComponent ci,
                        unsigned long offset,
                        unsigned long limit);
```

`ci`	Specifies the movie import component for the request. Applications obtain this reference from the Component Manager's `OpenComponent` function.
`offset`	A byte offset into the file that indicates where the import operation begins.
`limit`	A byte offset into the file that indicates the last data in the file that can be imported.

DISCUSSION

The `MovieImportSetOffsetAndLimit` function is especially useful when one file format is embedded in another; it allows your application to skip header data for the enclosing file and begin importing data at the start of the desired format.

Not all movie import components support the `MovieImportSetOffsetAndLimit` function. Those that do include the movie import components for the `kQTFileTypeAIFF`, `kQTFileTypeWave`, and `kQTFileMuLaw` file types. Those that do not return the `badComponentSelector` result code in response to a `MovieImportSetOffsetAndLimit` call.

RESULT Codes

`badComponentInstance`	0x80008001	Invalid movie import component instance
`badComponentSelector`	0x80008002	Function not supported

MovieImportGetSettingsAsAtomContainer

The `MovieImportGetSettingsAsAtomContainer` routine retrieves the current settings from the movie import component.

```
pascal ComponentResult  MovieImportGetSettingsAsAtomContainer (
                    MovieImportComponent  ci,
                    QTAtomContainer *settings);
```

`ci`	The movie import component instance.
`settings`	The address where the reference to the newly created atom container should be stored by the call.

DISCUSSION

The caller is responsible for disposing of the returned QT atom container.

MovieImportSetSettingsFromAtomContainer

The `MovieImportSetSettingsFromAtomContainer` routine sets the movie import component's current configuration from the passed settings data.

```
pascal ComponentResult MovieImportSetSettingsFromAtomContainer (
                    MovieImportComponent  ci,
                    QTAtomContainer  settings);
```

`ci`	The movie import component instance.
`settings`	Contains a QT atom container reference to the settings.

DISCUSSION

The settings QT atom container may contain atoms other than those expected by the particular component type or may be missing certain atoms. The function uses only those settings it understands.

Exporting Movie Data

This section describes new functions provided by movie data export components.

Since QuickTime 1.6.1, the sound movie export component has been updated to take advantage of Macintosh Sound Manager 3.0. Previously, only the first sound track in the movie was exported. Now sound tracks are mixed together before they are exported. If you want to use sound mixing, you can use the `PutMovieIntoTypedHandle` function to take advantage of the export component. Furthermore, you can now specify the format of the exported sound, so you can convert 16-bit sound to 8-bit sound or reduce stereo to monaural. See *Inside Macintosh: QuickTime* for a description of the `PutMovieIntoTypedHandle` function.

MovieExportGetAuxillaryData

QuickTime 1.6.1 added a new result code to this function. A movie export component returns the following result code when `MovieExportGetAuxillaryData` is called requesting a type of auxiliary data that the component cannot generate.

RESULT CODES

`auxillaryExportDataUnavailable`	-2058	Cannot generate the requested type of auxiliary data.

MovieExportSetSampleDescription

The `MovieExportSetSampleDescription` function allows your application to request the format of the exported data. This function is supported by the sound movie export component, for example.

```
pascal ComponentResult MovieExportSetSampleDescription(
                        MovieExportComponent ci,
                        SampleDescriptionHandle desc,
                        OSType mediaType);
```

`ci`	Specifies the movie component for the request. Applications obtain this reference from the Component Manager's `OpenComponent` function.
`desc`	Contains a handle to a valid QuickTime sample description.
`mediaType`	Indicates the type of media the sample description is for. For example, if the sample description was a sound description, this parameter would be set to `SoundMediaType`.

DISCUSSION

A movie export component may use all, some, or none of the settings from the sample description.

RESULT Codes

`badComponentInstance`	0x80008001	Invalid movie export component instance
`badComponentSelector`	0x80008002	Function not supported

MovieExportValidate

The MovieExportValidate function allows your application to determine whether a movie export component can export all the data for a specified movie or track.

```
pascal ComponentResult MovieExportValidate(
                         MovieExportComponent ci,
                         Movie theMovie,
                         Track onlyThisTrack,
                         Boolean *valid);
```

ci	Specifies the movie export component for the request. Applications obtain this reference from the Component Manager’s OpenComponent function.
theMovie	Specifies the movie to validate.
onlyThisTrack	Specifies a track within the movie to validate, or nil if the entire movie is to be validated.
valid	A pointer to a Boolean value. If the data for the movie or track can be exported by the component, the value is true.

DISCUSSION

MovieExportValidate allows an application to determine if a particular movie or track could be exported by the specified movie data export component. The movie or track is passed in the theMovie and onlyThisTrack parameters as they are passed to MovieExportToFile.

Although a movie export component can export one or more media types, it may not be able to export all the kinds of data stored in those media. The MovieExportValidate function allows applications to get this additional information.

Movie data export components that implement this function also set the canMovieExportValidateMovie flag in their component flags.

RESULT Codes

`badComponentInstance`	0x80008001	Invalid movie export component instance
`badComponentSelector`	0x80008002	Function not supported

Exporting Data from Sources Other Than Movies

This section describes new functions provided by movie data export components that allow applications to export data from sources other than movies.

MovieExportAddDataSource

The function allows you to define a data source for use with an export operation performed by `MovieExportFromProceduresToDataRef`.

```
pascal ComponentResult MovieExportAddDataSource (MovieExportComponent ci,
                     OSType trackType, TimeScale scale, long *trackID,
                     MovieExportGetPropertyUPP getPropertyProc,
                     MovieExportGetDataUPP getDataProc, void *refCon);
```

`ci`	Identifies the Movie Toolbox's connection to your component.
`trackType`	Specifies the type of media provided by this data source. This normally corresponds to a QuickTime media type such as `VideoMediaType` or `SoundMediaType`.
`scale`	Specifies the time scale for time values passed to `getDataProc` parameter. If the source data is being taken from a QuickTime track, this value is typically the media's `TimeScale`.
`trackID`	Contains an identifier for the data source. This identifier is returned from the call.
`getPropertyProc`	Specifies a callback function that provides information about processing source samples.
`getDataProc`	Specifies a callback function the export component uses to request sample data.

`refCon`	Passed to the procedures specified in the `getPropertyProc` and `getDataProc` parameters.

DISCUSSION

Before starting an export operation, all the data sources must be defined by calling `MovieExportAddDataSource` once for each data source.

MovieExportFromProceduresToDataRef

The `MovieExportFromProceduresToDataRef` function exports data provided by `MovieExportAddDataSource` to a location specified by `dataRef` and `dataRefType`.

```
pascal ComponentResult MovieExportFromProceduresToDataRef
                        (MovieExportComponent ci, Handle dataRef,
                        OSType dataRefType);
```

ci	Specifies the movie component for the export operation.
`dataRef`	Specifies the data reference for the export operation.
`dataRefType`	Specifies the type identifier for the data reference specified by `dataRef`.

DISCUSSION

Movie data export components that support export operations from procedures must set the `canMovieExportFromProcedures` flag in their component flags.

Typically `dataRef` contains a Macintosh file alias and `dataRefType` is set to `rAliasType`.

MovieExportToDataRef

`MovieExportToDataRef` allows an application to request that data be exported to a data reference, instead of to a file.

```
pascal ComponentResult MovieExportToDataRef(MovieExportComponent ci,
                     Handle dataRef, OSType dataRefType, Movie theMovie,
                     Track onlyThisTrack, TimeValue startTime,
                     TimeValue duration);
```

`ci`	Identifies the Movie Toolbox's connection to your component.
`dataRef`	Contains a handle to a data reference indicating where the data should be stored.
`dataRefType`	Specifies the type of the data reference. For exporting to a file, the `dataRef` is a Macintosh file alias and the `dataRefType` is `rAliasType`.
`theMovie`	Identifies the movie for this operation. This movie identifier is supplied by the toolbox. Your component may use this identifier to obtain sample data from the movie or to obtain information about the movie.
`onlyThisTrack`	Identifies a track that is to be converted. This track identifier is supplied by the toolbox. If this parameter contains a track identifier, your component must convert only the specified track.
`startTime`	Specifies the starting point of the track or movie segment to be converted. This time value is expressed in the movie's time coordinate system.
`duration`	Specifies the duration of the track or movie segment to be converted. This duration value is expressed in the movie's time coordinate system.

MovieExportGetSettingsAsAtomContainer

The `MovieExportGetSettingsAsAtomContainer` routine retrieves the current settings from the movie export component.

```
pascal ComponentResult MovieExportGetSettingsAsAtomContainer (
                    MovieExportComponent ci,
                    QTAtomContainer *settings);
```

`ci`	The movie export component instance.
`settings`	The address where the newly-created atom container should be stored by the call.

DISCUSSION

The caller is responsible for disposing of the returned QT atom container.

MovieExportSetSettingsFromAtomContainer

The `MovieExportSetSettingsFromAtomContainer` routine sets the movie export component's current configuration from the passed settings data.

```
pascal ComponentResult MovieExportSetSettingsFromAtomContainer (
                    MovieExportComponent  ci,
                    QTAtomContainer settings);
```

`ci`	The movie export component instance.
`settings`	This contains a QT atom container reference to the settings.

DISCUSSION

The settings QT atom container may contain atoms other than those expected by the particular component type or may be missing certain atoms. The `MovieExportSetSettingsFromAtomContainer` routine uses only those settings it understands.

Functions Provided by the Movie Export Component

This section documents routines provided by the QuickTime movie export component. These functions can be used by other data export components to implement their own functionality.

MovieExportNewGetDataAndPropertiesProcs

`MovieExportNewGetDataAndPropertiesProcs` returns `MovieExportGetProperty` and `MovieExportGetData` procedures that can be passed to `MovieExportAddDataSource` to create a new data source. This function exists in order to provide a standard way of getting data using this protocol out of a movie or track.

```
pascal ComponentResult MovieExportNewGetDataAndPropertiesProcs
                     (MovieExportComponent ci, OSType trackType,
                     TimeScale *scale, Movie theMovie, Track theTrack,
                     TimeValue startTime, TimeValue duration,
                     MovieExportGetPropertyUPP *getPropertyProc,
                     MovieExportGetDataUPP *getDataProc, void **refCon);
```

`ci`	Identifies the Movie Toolbox's connection to your component.
`trackType`	Specifies the format of the data to be generated by the returned `getDataProc`.
`scale`	The time scale is returned from this function, and should be passed on to `MovieExportAddDataSource` with the procedures.
`theMovie`	Identifies the movie for this operation. This movie identifier is supplied by the toolbox. Your component may use this identifier to obtain sample data from the movie or to obtain information about the movie.
`theTrack`	Identifies the track for this operation. This track identifier is supplied by the toolbox.
`startTime`	Specifies the starting point of the track or movie segment to be converted. This time value is expressed in the movie's time coordinate system.

`duration` Specifies the duration of the track or movie segment to be converted. This duration value is expressed in the movie's time coordinate system.

`getPropertyProc`
Specifies a callback function that provides information about processing source samples.

`getDataProc` Specifies a callback function the export component uses to request sample data.

`refCon` Passed to the procedures specified in the `getPropertyProc` and `getDataProc` parameters.

DISCUSSION

This function is only implemented by movie data export components. The returned procedures must be disposed by calling `MovieExportDisposeGetDataAndPropertiesProc`.

MovieExportDisposeGetDataAndPropertiesProcs

Disposes of the memory associated with the procedures returned by `MovieExportNewGetDataAndPropertiesProcs`.

```
pascal ComponentResult MovieExportDisposeGetDataAndPropertiesProcs
                    (MovieExportComponent ci,
                    MovieExportGetPropertyUPP getPropertyProc,
                    MovieExportGetDataUPP getDataProc, void *refCon);
```

`ci` Identifies the Movie Toolbox's connection to your component.

`getPropertyProc`
Specifies a callback function that provides information about processing source samples.

`getDataProc` Specifies a callback function the export component uses to request sample data.

`refCon` Passed to the procedures specified in the `getPropertyProc` and `getDataProc` parameters.

Application-Defined Functions

The `MovieExportGetPropertyProcPtr` and `MovieExportGetDataProcPtr` routines are closely associated with `MovieExportFromProcedures`.

MovieExportGetPropertyProcPtr

You use `MovieExportGetPropertyProcPtr` to return parameters that determine the appropriate format for the output data.

```
typedef pascal OSErr (*MovieExportGetPropertyProcPtr)(void *refcon,
                     long trackID, OSType propertyType,
                     void *propertyValue);
```

`refCon`	Contains the value passed to `MovieExportAddDataSource` in the `refCon` parameter.
`trackID`	Specifies the value returned from `MovieExportAddDataSource`.
`propertyValue`	Contains a pointer to the location of the requested property information.
`propertyType`	Specifies the property being requested. All currently defined property types are listed in Table 11-1.

Table 11-1 Currently supported property types

Property	**Type**	**Track Type**
`scSoundSampleRateType`	`UnsignedFixed`	`sound`
`scSoundSampleSizeType`	`short`	`sound`
`scSoundChannelCountType`	`short`	`sound`
`scSoundCompressionType`	`OSType`	`sound`
`meWidth`	`Fixed`	`video`
`meHeight`	`Fixed`	`video`
`scSpatialSettingsType`	`SCSpatialSettings`	`video`
`scTemporalSettingsType`	`SCTemporalSettings`	`video`
`scDataRateSettingsType`	`SCDataRateSettings`	`video`

DISCUSSION

The function defined by `MovieExportGetPropertyProcPtr` type is passed to `MovieExportAddDataSource` to define a new data source for an export operation. For example, a video export operation may call the function to determine the dimensions of the destination video track.

If the function doesn't have a setting for a requested property, it should return an error. The export component provides a default value for the property based on the source data format. For example, if no values for video track width and height properties were provided by the callback function, the dimensions of the source data would be used.

MovieExportGetDataProcPtr

You use the function defined by `MovieExportGetDataProcPtr` to define a data source for an export operation.

```
typedef pascal OSErr (*MovieExportGetDataProcPtr)(void *refCon,
                     MovieExportGetDataParams *params);
```

The sample request is made through a parameter block. The data structure used is shown below.

```
struct MovieExportGetDataParams {
    long                        recordSize;
    long                        trackID;
    TimeScale                   sourceTimeScale;
    TimeValue                   requestedTime;
    TimeValue                   actualTime;
    Ptr                         dataPtr;
    long                        dataSize;
    SampleDescriptionHandle     desc;
    OSType                      descType;
    long                        descSeed;
    long                        requestedSampleCount;
    long                        actualSampleCount;
    TimeValue                   durationPerSample;
    long                        sampleFlags;
};
```

```
typedef struct MovieExportGetDataParams MovieExportGetDataParams;
```

`refCon`	Contains the value passed to `MovieExportAddDataSource` in the `refCon` parameter.
`recordSize`	Contains the total size in bytes of the `MovieExportGetDataParams` structure. This is provided to allow for additional parameters to be added safely in the future.
`trackID`	Specifies the data source. The `trackID` is returned when the data source is added by calling `MovieExportAddDataSource`.
`sourceTimeScale`	Specifies the time scale for this data source. This value is the same time scale that is passed to `MovieExportAddDataSource`.
`requestedTime`	Specifies the time of the media requested by the exporter. The time scale is the same one specified when adding a data source with `MovieExportAddDataSource`.
`actualTime`	Specifies the time actually referred to by the returned media data. This value is provided by `MovieExportGetDataProcPtr`, and is usually the same as `requestedTime`.
`dataPtr`	Contains a 32-bit pointer to the media data.
`dataSize`	Specifies the size in bytes of the data pointed to by `dataPtr`.
`desc`	Contains a `SampleDescriptionHandle` describing the format of the data pointed to by `dataPtr`. For video data, this is an `ImageDescriptionHandle`. For sound data, this is a `SoundDescriptionHandle`.
`descType`	Specifies the type of `SampleDescriptionHandle`. For example, if `SampleDescriptionHandle` is `ImageDescriptionHandle, descType` is set to `VideoMediaType`.
`descSeed`	Specifies which `SampleDescriptionHandle` represented by the current value of `desc`. Some data sources contain different kinds of data at different times. For example, a video data source may contain both JPEG and uncompressed raw data. Whenever the data source switches from one type of data to another, change `descSeed` to notify the exporter. In the case of an export operation that is providing its source data from a QuickTime movie track, `descSeed` is equal to the sample description index of the sample being returned.

`requestedSampleCount`	Specifies the number of samples the exporter can work with. The function can return more or fewer samples than requested. For video, this value is always 1.
`actualSampleCount`	Specifies the number of samples actually returned. The function must fill in this field.
`durationPerSample`	Specifies the duration of every sample returned. For sound data, `durationPerSample` always contains 1. For video data, `durationPerSample` contains the duration of the returned sample, expressed in the time scale defined when the data source was created.
`sampleFlags`	Contains the flags for the returned samples. The only defined flag is `mediaSampleNotSync`, which is usually only returned for frame-differenced video sample data. The function must fill in this field.

DISCUSSION

The function defined by the `MovieExportGetDataProcPtr` type is passed to `MovieExportAddDataSource` to define a new data source for an export operation. The function is used by the exporting application to request source media data to be used in the export operation. For example, in a video export operation, frames of video data (either compressed or uncompressed) are provided. In a sound export operation, buffers of audio (either compressed or uncompressed) are provided.

The data pointed to by `dataPtr` must remain valid until the next call to this `MovieExportGetDataProcPtr` function. The `MovieExportGetDataProcPtr` function is responsible for allocating and disposing of the memory associated with this data pointer.

Derived Media Handler Components

This chapter discusses the changes to derived media handler components as documented in Chapter 10 of *Inside Macintosh: QuickTime Components*.

Derived Media Handler Components Reference

Constants

Media Video Parameters

The `whichparam` parameter to the `MediaSetVideoParam` and `MediaGetVideoParam` functions specifies which video parameter you want to adjust. QuickTime defines these constants that you can use to configure the `whichparam` parameter.

```
enum {
    kMediaVideoParamBrightness  = 1,
    kMediaVideoParamContrast    = 2,
    kMediaVideoParamHue         = 3,
    kMediaVideoParamSharpness   = 4,
    kMediaVideoParamSaturation  = 5,
    kMediaVideoParamBlackLevel  = 6,
    kMediaVideoParamWhiteLevel  = 7
};
```

Constant descriptions

`kMediaVideoParamBrightness`
: The brightness value controls the overall brightness of the digitized video image. Brightness values range from 0 to 65,535, where 0 is the darkest possible setting and 65,535 is the lightest possible setting.

`kMediaVideoParamContrast`
: The contrast value ranges from 0 to 65,535, where 0 represents no change to the basic image and larger values increase the contrast of the video image (that is, increase the slope of the transform).

`kMediaVideoParamHue`
: Hue is similar to the tint control on a television. It is specified in degrees with complementary colors set 180 degrees apart (red is 0°, green is +120°, and blue is –120°). QuickTime supports 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.

`kMediaVideoParamSharpness`
: The sharpness value ranges from 0 to 65,535, where 0 represents no sharpness filtering and 65,535 represents full sharpness filtering. Higher values result in a visual impression of increased picture sharpness

`kMediaVideoParamSaturation`
: The saturation value 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.

`kMediaVideoParamBlackLevel`
: Black level refers to the degree of blackness in an image.The highest setting produces an all-black image; on the other hand, the lowest setting yields little, if any, black even with black objects in the scene. Black level values range from 0 to 65,535, where 0 represents the maximum black value and 65,535 represents the minimum black value.

`kMediaVideoParamWhiteLevel`
: White level refers to the degree of whiteness in an image.

White level values range from 0 to 65,535, where 0 represents the minimum white value and 65,535 represents the maximum white value

Data Types

The `GetMovieCompleteParams` data type, which defines the layout of the complete movie parameter structure used by the `MediaInitialize` function, has a new parameter, `inputMap`, which is a reference to the media's input map. The media input map should not be modified or disposed. Because of this change, the `version` field of the `GetMovieCompleteParams` data type has been changed from 0 to 1.

Derived Media Handler Component Functions

Managing Your Media Handler Component

This section describes functions that apply to all derived media handler components.

MediaIdle

There is a minor change to the `MediaIdle` function that is related to the new media handler support for partial screen redrawing.

From time to time, your derived media handler component may determine that only a portion of the available drawing area needs to be redrawn. You can signal that condition to the base media handler component by setting the `mPartialDraw` flag to 1 in the flags your component returns to the Movie Toolbox from your `MediaIdle` function. You return these flags using the `flagsOut` parameter.

Whenever you set this flag to 1, the toolbox calls your component's `MediaGetDrawingRgn` function in order to determine the portion of the image that needs to be redrawn.

As an example, consider a full-screen animation. Only rarely is the entire image in motion. Typically, only a small portion of the screen image moves. By using partial redrawing, you can significantly improve the playback performance of such a movie.

See also: `MediaInvalidateRegion` (page 403) and `MediaGetDrawingRgn` (page 417).

General Data Management

This section contains several new functions to support modifier tracks, active segments, and video settings. These functions apply to all derived media handler components.

MediaGSetActiveSegment

The `MediaGSetActiveSegment` function informs your derived media handlers of the current active segment.

```
pascal ComponentResult MediaGSetActiveSegment(
                        MediaHandler mh,
                        TimeValue activeStart,
                        TimeValue activeDuration);
```

`mh`	Identifies the Movie Toolbox's connection to your derived media handler.
`activeStart`	Contains the starting time of the active segment to play. This time value is expressed in your movie's time scale.
`activeDuration`	Contains a time value that specifies the duration of the active segment. This value is expressed in the movie's time scale.

DISCUSSION

Using the `SetMovieActiveSegment` function, an application can limit the time segment of the movie that will be used for play back. Derived media handlers are given the values for the active segment by the `MediaGSetActiveSegment` function called by the toolbox. Active segment information is usually only needed by media handlers that perform their own scheduling.

MediaInvalidateRegion

The `MediaInvalidateRegion` function updates the invalidated display region the next time `MediaIdle` is called.

```
pascal ComponentResult MediaInvalidateRegion(
                        MediaHandler mh,
                        RgnHandle invalRgn);
```

`mh`	Identifies the Movie Toolbox's connection to your derived media handler.
`invalRgn`	Contains a handle to a region that has been invalidated. Your media handler should not dispose or modify this region. The `invalRgn` parameter is never `nil`.

DISCUSSION

The `MediaInvalidateRegion` function is called by the toolbox when `UpdateMovie` or `InvalidateMovieRegion` is called with a region that intersects your media's track.

Derived media handlers need to implement `MediaInvalidateRegion` only if they can perform efficient updates on a portion of their display area.

If a media handler implements the `MediaInvalidateRegion` function, it is responsible for ensuring that the appropriate areas of the screen are updated on the next call to `MediaIdle`. If a media handler does not implement this function, the base media handler sets the `mMustDrawflag` the next time `MediaIdle` is called.

MediaGetNextStepTime

The `MediaGetNextStepTime` function searches for the next forward or backward **step time** from the given media time. The step time is the time of the next frame. This function allows a derived media handler to return the next step time from the specified media time.

```
pascal ComponentResult MediaGetNextStepTime(
                     MediaHandler mh,
                     short flags,
                     TimeValue mediaTimeIn,
                     TimeValue *mediaTimeOut,
                     Fixed rate);
```

`mh`	Identifies the Movie Toolbox's connection to your derived media handler.
`flags`	The following `interestingTimeFlags` flags are defined:

`nextTimeStep`	Searches for the next frame in the media. Set this flag to 1 to search for the next frame.
`nextTimeEdgeOk`	Instructs the toolbox that you are willing to receive information about elements that begin or end at the time specified by the `mediaTimeIn` 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.

`mediaTimeIn`	Specifies a time value that establishes the starting point for the search. This time value is in the media's time scale.
`mediaTimeOut`	The step time calculated by the media handler. The media handler should return 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.
`rate`	Contains the search direction. Negative values search backward from the starting point specified in the `mediaTimeIn` parameter. Other values cause a forward search.

DISCUSSION

The mechanism in QuickTime used for stepping backwards and forwards a frame at a time are the interesting time calls: `GetMovieNextInterestingTime`, `GetTrackNextInterestingTime`, and `GetMediaNextInterestingTime`. The normal method for stepping forward to the next frame is to use one of these calls to locate the time of the next visual sample. This works well for most media types, including video and text. Unfortunately, it does not work well for MPEG. QuickTime stores an entire MPEG stream as a single sample. Therefore stepping to the next sample would actually skip to the end of the entire sequence. To solve this problem, QuickTime 2.1 introduced a new flag, `nextTimeStep`, for the interesting time calls. This flag is specifically intended for stepping through frames.

The standard QuickTime movie controller has been updated to use this flag in place of the old `nextTimeSample` flag. The `nextTimeStep` flag works correctly for all media types including video and MPEG. To work correctly with MPEG, applications that implement stepping functionality should use this new flag.

See *Inside Macintosh: QuickTime* for more information on interesting times.

MediaTrackReferencesChanged

The `MediaTrackReferencesChanged` function notifies the derived media handler whenever the track references in the movie change.

```
pascal ComponentResult MediaTrackReferencesChanged (MediaHandler mh);
```

`mh`	Identifies the Movie Toolbox's connection to your derived media handler.

DISCUSSION

When an application creates, modifies, or deletes a track reference, the media handler's `MediaTrackReferencesChanged` function is called. When this function is called, a media handler should rebuild all information about track references and reset its values for all media inputs to their default values.

MediaTrackPropertyAtomChanged

The `MediaTrackPropertyAtomChanged` function notifies the derived media handler whenever its media property atom has changed.

```
pascal ComponentResult MediaTrackPropertyAtomChanged (MediaHandler mh);
```

`mh`	Identifies the Movie Toolbox's connection to your derived media handler.

DISCUSSION

The `MediaTrackPropertyAtomChanged` function is called whenever `SetMediaPropertyAtom` is called. If the media handler uses information from the property atom, it should rebuild the information at this time.

MediaSetTrackInputMapReference

When an application modifies the media input map, the `MediaSetTrackInputMap` function provides the derived media handler with the updated input map.

```
pascal ComponentResult MediaSetTrackInputMapReference(
                        MediaHandler mh,
                        QTAtomContainer inputMap);
```

`mh`	Identifies the Movie Toolbox's connection to your derived media handler.
`inputMap`	Specifies the media input map for this operation. Do not modify or dispose of the input map provided.

DISCUSSION

When the `MediaSetTrackInputMapReference` function is called, the media handler should store the updated input map and recheck the types of all inputs, if it is caching this information. The input map reference passed to this function should not be disposed of or modified by the media handler.

MediaGetSampleDataPointer

The `MediaGetSampleDataPointer` function allows a derived media handler to obtain a pointer to the sample data for a particular sample number, the size of that sample, and the index of the sample description associated with that sample.

```
pascal ComponentResult MediaGetSampleDataPointer(
                     MediaHandler mh,
                     long sampleNum,
                     Ptr *dataPtr,
                     long *dataSize,
                     long *sampleDescIndex);
```

`mh`	Identifies the Movie Toolbox's connection to your derived media handler.
`sampleNum`	Contains the number of the sample that is to be loaded.
`dataPtr`	Contains a pointer to a pointer to receive the address of the loaded sample data.
`dataSize`	Contains a pointer to a field that is to receive the size, in bytes, of the sample.
`sampleDescIndex`	Contains a pointer to a long integer. The `MediaGetSampleDataPointer` 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`.

DISCUSSION

The `MediaGetSampleDataPointer` function returns a pointer to the data for a particular sample number from a movie data file.

This function provides access to the base media handler's caching services for sample data. It is a service provided by the base media handler for its clients.

Each call to `MediaGetSampleDataPointer` must be balanced by a call to `MediaReleaseSampleDataPointer` or the memory will not be released.

`MediaGetSampleDataPointer` generally provides better overall performance than `GetMediaSample`.

MediaReleaseSampleDataPointer

The `MediaReleaseSampleDataPointer` function balances calls to `MediaGetSampleDataPointer` to release the memory. This function should be used only by derived media handlers.

```
pascal ComponentResult MediaReleaseSampleDataPointer(
                     MediaHandler mh,
                     long sampleNum);
```

`mh`	Identifies the Movie Toolbox's connection to your derived media handler.
`sampleNum`	Contains the number of the sample that is to be released.

MediaCompare

The `MediaCompare` function allows a media handler to determine whether the Movie Toolbox should allow one track to be pasted into another. `MediaCompare` is provided with a reference to the media with which it should be compared.

```
pascal ComponentResult MediaCompare(
                     MediaHandler mh,
                     Boolean *isOK,
                     Media srcMedia,
                     ComponentInstance srcMediaComponent);
```

`mh`	Identifies the toolbox's connection to your derived media handler.

`isOK`	Contains a pointer to a Boolean value. Your media handler must set this Boolean value to indicate whether the source media and the media associated with the media handler have equivalent media settings, so that pasting the two together would cause no media information loss.
`srcMedia`	Specifies the source media for this operation.
`srcMediaComponent`	Specifies the source media component for this operation.

MediaSetVideoParam

The `MediaSetVideoParam` function enables you to dynamically adjust the brightness, contrast, hue, sharpness, saturation, black level, and white level of a video image.

```
pascal ComponentResult MediaSetVideoParam(
                      MediaHandler mh,
                      long whichParam,
                      unsigned short *value);
```

`mh`	Identifies the Movie Toolbox's connection to your derived media handler.
`whichParam`	Contains a long integer that specifies the number of the video parameter that should be adjusted.
`value`	Contains the actual value of the video parameter. The meaning of the values vary depending on the implementation.

DISCUSSION

The `MediaSetVideoParam` and `MediaGetVideoParam` functions are currently used by the MPEG media handler.

See “Media Video Parameters” (page 399) for the constants and values you can use for the `whichParam` and `value` parameters.

MediaGetVideoParam

The `MediaGetVideoParam` function enables you to retrieve the value of the brightness, contrast, hue, sharpness, saturation, black level, or white level of a video image.

```
pascal ComponentResult MediaGetVideoParam(
                          MediaHandler mh,
                          long whichParam,
                          unsigned short *value);
```

`mh`	Identifies the Movie Toolbox's connection to your derived media handler.
`whichParam`	Contains a long integer thatspecifies the number of the video parameter whose value you want to retrieve.
`value`	Contains the actual value of the requested video parameter. The meaning of the values vary depending on the implementation.

DISCUSSION

The `MediaSetVideoParam` and `MediaGetVideoParam` functions are currently used by the MPEG media handler.

See the *Inside Macintosh: QuickTime Components* chapter on Video Digitizer Components for more information about the `whichParam` and `value` parameters.

MediaSetNonPrimarySourceData

The `MediaSetNonPrimarySourceData` function allows a media handler to support receiving media data from other media handlers.

```
pascal ComponentResult MediaSetNonPrimarySourceData(
                          MediaHandler mh,
                          long inputIndex,
                          long dataDescriptionSeed,
                          Handle dataDescription,
                          void *data,
                          long dataSize,
                          ICMCompletionProcRecordPtr asyncCompletionProc,
                          UniversalProcPtr transferProc,
                          void *refCon);
```

`mh`	Identifies the Movie Toolbox's connection to your derived media handler.
`inputIndex`	This value is the ID of the entry in the media's input map to which the data provided by the call corresponds.
`dataDescriptionSeed`	This value is changed each time the `dataDescription` has changed. This allows for a quick check by the media handler to see if the `dataDescription` has changed.
`dataDescription`	A handle to a data structure describing the input data.
`data`	Points to the input data. This pointer must contain a 32-bit address.
`dataSize`	Contains the size of the sample in bytes.
`asyncCompletionProc`	Points to a completion function structure. If the `asyncCompletionProc` is set to `nil`, the data pointer will be valid only for the duration of this call. If the `asyncCompletionProc` is not `nil`, it contains an `ICMCompletionProcRecord` that must be called when your media handler is done with the provided data pointer.
`transferProc`	A routine that allows the application to transform the type of the input data to the kind of data preferred by the codec. The client of the codec passes the source data in the form most convenient for it. If the codec needs the data in another form, it can negotiate with the caller or directly with the Image Compression Manager to obtain the required data format.
`refCon`	Contains a reference constant (defined as a `void` pointer). Your application specifies the value of this reference constant in the function structure you pass to the media handler.

DISCUSSION

There are two tasks required to support modifier tracks in a derived media handler: sending and receiving. The base media handler takes care of sending data for all its clients. Therefore, authors of derived media handlers do not usually need to implement sending data support.

Receiving data is a more complex situation. The base media handler takes care of input types that it understands. The base media handler supports the following types of data:

```
kTrackModifierTypeMatrix
kTrackModifierTypeGraphicsMode
kTrackModifierTypeClip
kTrackModifierTypeVolume
kTrackModifierTypeBalance
```

If a media handler wants to support receiving other types of data it must implement the `MediaSetNonPrimarySourceData` routine. `MediaSetNonPrimarySourceData` is called by modified tracks to supply the current data for each input. All unrecognized input types should be delegated to the base media handler so that they can be handled.

The following is a basic shell implementation of a derived media handler's `MediaSetNonPrimarySourceData` function. Note that your derived media handler must delegate all input types it does not handle to the base media handler.

```
pascal ComponentResult MySetNonPrimarySourceData( MyGlobals store,
    long inputIndex, long dataDescriptionSeed, Handle dataDescription,
    void *data, long dataSize,
    ICMCompletionProcRecordPtr asyncCompletionProc,
    UniversalProcPtr transferProc, void *refCon )
{
ComponentResult err = noErr;
QTAtom inputAtom;
QTAtom typesAtom;
long inputType;

// determine what kind of input this is
inputAtom = QTFindChildByID(store->inputMap,
    kParentAtomIsContainer, kTrackModifierInput, inputIndex, nil);
if (!inputAtom) {
    err = cannotFindAtomErr;
    goto bail;
}
    typesAtom = QTFindChildByID(store->inputMap, inputAtom,
    kTrackModifierType, 1, nil);
err = QTCopyAtomDataToPtr(store->inputMap, typesAtom, false,
    sizeof(inputType), &inputType, nil);
if (err) goto bail;

switch(inputType) {
    case kMyInputType:
        if (data == nil) {
            // no data, reset to default value
        }
        else {
            // use this data
            // when done, notify caller we're done with this data
            if (asyncCompletionProc)
                CallICMCompletionProc(
                    asyncCompletionProc->completionProc,
                    noErr, codecCompletionSource | codecCompletionDest,
                    asyncCompletionProc->completionRefCon);
        }
        break;

    default:
        err = MediaSetNonPrimarySourceData(store->delegateComponent,
            inputIndex, dataDescriptionSeed, dataDescription, data,
            dataSize, asyncCompletionProc, transferProc, refCon);
        break;
}
bail:
    return err;
}
```

MediaGetOffscreenBufferSize

The `MediaGetOffscreenBufferSize` function determines the dimensions of the offscreen buffer.

```
pascal ComponentResult MediaGetOffscreenBufferSize(
                     MediaHandler mh,
                     Rect *bounds,
                     short depth,
                     CTabHandle ctab);
```

`mh`	Identifies the Movie Toolbox's connection to your derived media handler.
`bounds`	Specifies the boundaries of your offscreen buffer.
`depth`	Specifies the depth of the offscreen.
`ctab`	Contains a handle to the color table associated with the offscreen buffer.

DISCUSSION

Before the base media handler allocates an offscreen buffer for your derived media handler, it calls your `MediaGetOffscreenBufferSize` function. The depth and color table used for the buffer are also passed. When this function is called, the `bounds` parameter specifies the size that the base media handler intends to use for your offscreen buffer. You can modify this as appropriate before returning. This capability is useful if your media handler can draw only at particular sizes. It is also useful for implementing antialiased drawing; you can request a buffer that is larger than your destination area and have the base media handler scale the image down for you.

RESULT CODES

`badComponentInstance`	0x80008001	Invalid component instance specified

MediaSetHints

The `MediaSetHints` function implements the appropriate behavior for the various media hints such as scrub mode and high-quality mode.

```
pascal ComponentResult MediaSetHints(
                     MediaHandler mh,
                     long hints);
```

`mh`	Identifies the Movie Toolbox's connection to your derived media handler.
`hints`	Contains all hint bits that currently apply to the given media.

DISCUSSION

When an application calls `SetMoviePlayHints` or `SetMediaPlayHints`, your media handler's `MediaSetHints` routine is called.

RESULT CODES

`badComponentInstance`	0x80008001	Invalid component instance specified

MediaGetName

The `MediaGetName` function returns the name of the media type. For example, the video media handler returns the string `'video'`.

```
pascal ComponentResult MediaGetName(
                     MediaHandler mh,
                     Str255 name,
                     long requestedLanguage,
                     long *actualLanguage);
```

`mh`	Identifies the Movie Toolbox's connection to your derived media handler.
`name`	Specifies where to return the name of the media type.
`requestedLanguage`	Specifies the language in which you want the `name` returned. This value is a standard Mac OS region code.
`actualLanguage`	Specifies the actual language in which the `name` is returned. This value is a standard Mac OS region code.
function result	The name of the media type.

RESULT CODES

`badComponentInstance`	0x80008001	Invalid component instance specified

Graphics Data Management

This section describes functions for managing graphics data. These functions apply to all derived media handler components.

MediaGetDrawingRgn

The `MediaGetDrawingRgn` function allows your derived media handler component to specify a portion of the screen that must be redrawn. This region is defined in the movie's display coordinate system.

```
pascal ComponentResult MediaGetDrawingRgn (MediaHandler mh, RgnHandle
                      *partialRgn);
```

`mh`	Identifies the Movie Toolbox's connection to your derived media handler.
`partialRgn`	Points to a handle that defines the screen region to be redrawn. Note that your component is responsible for disposing of this region once drawing is complete. Since the base media handler will use this region during redrawing, it is best to dispose of it when your component is closed.

DISCUSSION

The Movie Toolbox calls this function in order to determine what part of the screen needs to be redrawn. By default, thetoolbox redraws the entire region that belongs to your component. If your component determines that only a portion of the screen has changed, and has indicated this to thetoolbox by setting the `mPartialDraw` flag to 1 in the `flagsOut` parameter of the `MediaIdle` function, the toolbox calls your component's `MediaGetDrawingRgn` function. Your component returns a region that defines the changed portion of the track's display region.

RESULT CODES

`badComponentSelector`	0x80008002	Function not supported

Memory Manager errors

MediaGetGraphicsMode

The `MediaGetGraphicsMode` function allows you to obtain the graphics mode and blend color values currently in use by any media handler.

```
pascal ComponentResult MediaGetGraphicsMode (
                     MediaHandler mh,
                     long *mode,
                     RGBColor *opColor);
```

`mh`	Identifies the Movie Toolbox's connection to your derived media handler.
`mode`	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 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 toolbox supplies this value to QuickDraw when you draw in `addPin`, `subPin`, `blend`, `transparent`, or `graphicsModeStraightAlphaBlend` mode.

RESULT CODES

Component Manager errors, as documented in *Mac OS For QuickTime Programmers*.

SEE ALSO

You can set the graphics mode and blend color of any media handler by calling the `MediaSetGraphicsMode` function, described below.

MediaSetGraphicsMode

The `MediaSetGraphicsMode` function allows you to set the graphics mode and blend color of any media handler.

```
pascal ComponentResult MediaSetGraphicsMode (
                        MediaHandler mh,
                        long mode,
                        const RGBColor *opColor);
```

`mh`	Identifies the Movie Toolbox's connection to your derived media handler.
`mode`	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`, `transparent`, or `graphicsModeStraightAlphaBlend` mode.

RESULT CODES

Component Manager errors, as documented in *Mac OS For QuickTime Programmers*.

SEE ALSO

You can retrieve the graphics mode and blend color currently in use by any media handler by calling the `MediaGetGraphicsMode` function, described above.

Sound Data Management

This section describes functions for managing sound data. These functions apply to all derived media handler components.

MediaSetSoundLocalizationData

If you are creating a media handler that plays sound and wish to support 3D sound capabilities, you need to implement the `MediaSetSoundLocalizationData` routine.

```
pascal ComponentResult MediaSetSoundLocalizationData (MediaHandler mh,
                          Handle data);
```

`mh`	Identifies the Movie Toolbox's connection to your derived media handler.
`data`	The data passed to your media handler, in the format of a sound sprockets `SSpLocalizationData` record.

Discussion

This routine is passed a handle containing the new `SSpLocalizationData` record to use. If the handle is `nil`, it indicates that no 3D sound effects should be used. If you implement this routine, and return `noErr` as the result, it is assumed that your media handle assumes responsibility for disposing of the data handle passed. If the implementation of this routine returns an error, the caller will dispose of the handle. This behavior is implemented to minimize the copying of the settings handle, making it easier for developers to implement the `MediaSetSoundLocalizationData` function.

The `MediaSetSoundLocalizationData` is called regardless of whether the 3D sound settings were set on the track using `SetTrackSoundLocalizationSettings` or via a modifier track mechanism.

Management of Progressive Downloads

This section describes a function for progressive downloads. These functions apply to all derived media handler components.

MediaMakeMediaTimeTable

When an application or other software calls any of the Movie Toolbox's functions to create a time table for progressive downloads, the base media handler calls your derived media handler's `MediaMakeMediaTimeTable` function to create the time table.

```
pascal ComponentResult MediaMakeMediaTimeTable (
      MediaHandler mh,
      long **offsets,
      TimeValue startTime,
      TimeValue endTime,
      TimeValue timeIncrement,
      short firstDataRefIndex,
      short lastDataRefIndex,
      long *RetDataRefSkew);
```

`mh`	Specifies the media handler to create the time table.
`offsets`	A handle to an unlocked relocatable memory block that is allocated by an application or other software when it calls the toolbox's `QTMovieNeedsTimeTable`, `GetMaxLoadedTimeInMovie`, `MakeTrackTimeTable`, or `MakeMediaTimeTable` function. Your derived media handler returns the time table for the media in this block. Your media handler has to resize the handle.
`startTime`	Specifies the first point of the media to be included in the time table. This time value is expressed in the media's time coordinate system.
`endTime`	Specifies the last point of the media to be included in the time table. This time value is expressed in the media's time coordinate system.
`timeIncrement`	The resolution of the time table. The values in a time table are for a points in the media, and these points are separated by the amount of time specified by this parameter. The time value is expressed in the media's time coordinate system.
`firstDataRefIndex`	Specifies the first in the range of data reference indexes you are querying.

`lastDataRefIndex`

Specifies the last in the range of data reference indexes you are querying.

`RetDataRefSkew`

Contains a pointer to the number of entries, i.e., the number of entries in the offset table per data reference.

DISCUSSION

The toolbox calls your derived media handler's `MediaMakeMediaTimeTable` function whenever an application or other software calls the toolbox's `QTMovieNeedsTimeTable`, `GetMaxLoadedTimeInMovie`, `MakeTrackTimeTable`, or `MakeMediaTimeTable` function. When an application or other software calls one of these functions, it allocates an unlocked relocatable memory block for the time table to be returned and passes a handle to it in the `offsets` parameter. Your derived media handler must resize the block to accommodate the time table it returns.

The time table your derived media handler returns is a two-dimensional array of long integers that is organized as follows:

- Each row in the table contains values for one data reference.
- The first column in the table contains values for the time in the media specified by the `startTime` parameter, and each subsequent column contains values for the point in the media that is later by the value specified by the `timeIncrement` parameter.
- Each long integer value in the table specifies the offset, in bytes, from the beginning of the data reference for that point in the media.

The number of columns in the table must be equal to (`endTime` - `startTime`) / `timeIncrement`, rounded up.

Your derived media handler must also return the offset to the next row of the time table, in long integers, in the `retdataRefSkew` parameter. Because of alignment issues, this value is not always equal to (`endTime` - `startTime`) / `timeIncrement`, rounded up.

Tween Media Handler

This chapter describes the **tween media handler**, which lets you use tween data stored in a tween track to modify the presentation of other tracks algorithmically. For information about creating and using tweens, see Chapter 25, “Tween Components and Native Tween Types.”

A QuickTime movie can include one or more **tween tracks**, which are modifier tracks that contain tween data. When a movie includes a tween track, the tween media handler invokes the tween component (or native QuickTime code) needed to process the tween data and delivers the resulting values to one or more other media handlers.

The chapter is divided into the following major sections:

- “About the Tween Media Handler” (page 423) introduces the tween media handler, tween tracks, and tween data.
- “Using the Tween Media Handler” (page 424) describes how to create and use tween tracks to modify other tracks.
- “Constants” (page 430) describes the constants used with the tween media handler.

About the Tween Media Handler

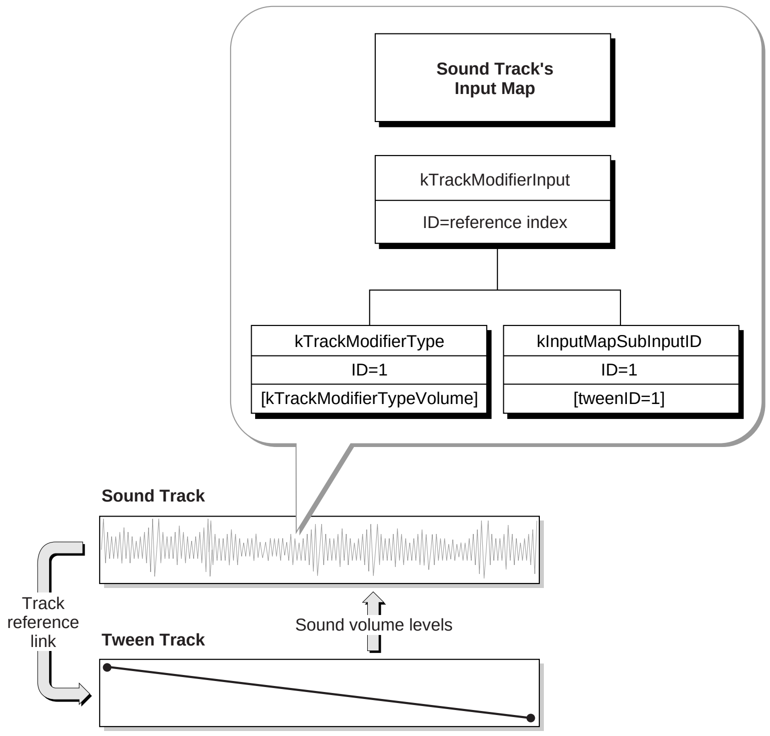
A **tween track** is a special track in a movie that is used exclusively as a modifier track. The data it contains, known as **tween data,** is used to generate values that modify the playback of other tracks, usually by interpolating values. The tween media handler sends these values to other media handlers; it never presents data. For an introduction to modifier tracks, see “Modifier Tracks” (page 44).

Typical tween components can just interpolate numeric values or can perform complex tweening, such as finding intermediate data between one matrix or polygon and another. For further information, see Chapter 25, “Tween Components and Native Tween Types.”

Using the Tween Media Handler

You can use the tween media handler to send tween values from a tween track to a receiving track, such as a video track or a sound track. To send tween values, you must create a tween track. The Movie Toolbox routes the data from the tween track to the receiving track based upon the receiving track’s input map, as shown in Figure 13-1.

Figure 13-1 Data routed from a tween track to a receiving track based on its input map



Creating a Tween Track

To create a tween track, you must:

- Create a tween track and its media.
- Create one or more tween media samples.
- Add the media samples to the tween media.
- Add the tween media to the track.
- Create a link from the tween track to the track to which the tween media handler should send tween values.
- Bind the tween entry to the desired attributes in the receiving track.

The sample code shown in this section creates a tween sample that interpolates a short integer from 255 to 0. The tween media is attached to the sound track of a QuickTime movie to modify the sound track's volume. Thus it creates a volume fadeout using the tween track. The data type for the tween component is `kTweenTypeShort`.

The sample code shown in Listing 13-1 creates a new track (`t`) to be used as the tween track and new tween media (type `TweenMediaType`).

Listing 13-1 Creating a tween track and tween media

```
Track t;
Media md;
SampleDescriptionHandle desc;

// ...
// set up the movie, m
// ...

// allocate a sample description handle
desc = (SampleDescriptionHandle)NewHandleClear (
    sizeof (SampleDescription));

// create the tween track, t
t = NewMovieTrack (m, 0, 0, kNoVolume);
```

```
// create the tween media, md
md = NewTrackMedia (t, TweenMediaType, 600, nil, 0);

(**desc).descSize = sizeof(SampleDescription);
```

Next, your application must create a tween media sample. The tween media sample is a QT atom container structure that contains one or more `kTweenEntry` atoms. (See Chapter 1, “Movie Toolbox,” for additional information on QT atom containers.) Each `kTweenEntry` atom defines a separate tween operation. A single tween sample can describe several parallel tween operations.

The sample code shown in Listing 13-2 creates a new QT atom container and inserts a `kTweenEntry` atom into the container. Then, it creates two leaf atoms, both children of the `kTweenEntry` atom. The first leaf atom (atom type `kTweenType`) contains the type of the tween data, in this case `kTweenTypeShort`. The second leaf atom (atom type `kTweenData`) contains the two data values for the tween operation, 512 and 0.

Remember that all data in QT atoms must be big-endian. The sample code shown in this chapter contains the endian conversion routines required for cross-platform compatibility.

Listing 13-2 Creating a tween sample

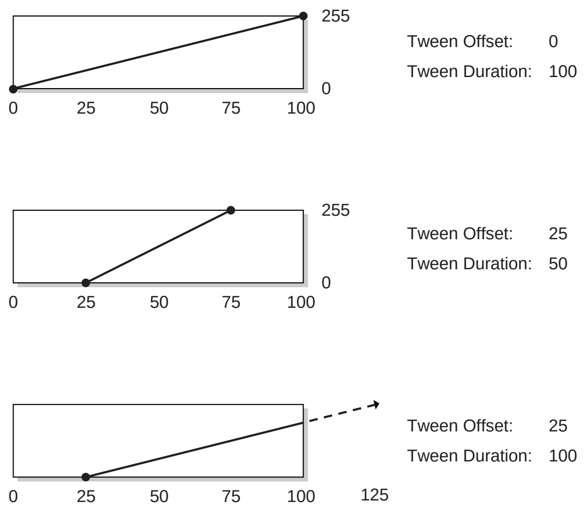
```
QTAtomContainer container = nil;
short tweenDataShort[2];
QTAtomType tweenType;

tweenDataShort[0] = EndianS16_NtoB(255);
tweenDataShort[1] = EndianS16_NtoB(0);

// create a new atom container to hold the sample
QTNewAtomContainer (&container);

// create the parent tween entry atom
QTInsertChild (container, kParentAtomIsContainer, kTweenEntry, 1, 0, 0,
    nil, &tweenAtom);

// add two child atoms to the tween entry atom
// * the type atom, kTweenType
tweenType = EndianU32_NtoB(kTweenTypeShort);
```

```
QTInsertChild (container, tweenAtom, kTweenType, 1, 0,
    sizeof(tweenType), &tweenType, nil);

// * the data atom, kTweenData
QTInsertChild (container, tweenAtom, kTweenData, 1, 0, sizeof(short) * 2,
    tweenDataShort, nil);
```

You do not have to start the tween at the beginning of the sample, nor do you have to stop at the end of the sample. You can specify the start of the tween and its duration by adding additional child atoms to the tween entry.

Figure 13-2 illustrates, for example, how you can use tween media to modify a sound track's volume. The first part of the illustration shows an example of tweening the sound volume from 0 to 255, with the tween offset at 0 and the tween duration at 100. In the second part, with the tween offset at 25 and the duration at 50, the tween has no effect until time 25—after which it causes the volume to fade in over the next 50 time units. The volume is left at 255. The third part shows the tween offset at 25 and the tween duration at 100. Since the offset plus the duration of this tween is greater than the duration of the tween media sample, the sound track never reaches full volume.

Figure 13-2 Using tween media to modify the sound track's volume



You can add a kTweenStartOffset atom to start the tween operation at 500 units into the sample with the following lines of code:

```
TimeValue time = EndianU32_NtoB(500);
QTInsertChild (container, tweenAtom, kTweenStartOffset, 1, 0,
    sizeof(TimeValue), &time, nil);
```

You can specify a duration for the tween operation independent of the sample's duration by adding a kTweenDuration atom to the tween entry, as follows:

```
TimeValue duration = EndianU32_NtoB(1000);
QTInsertChild (container, tweenAtom, kTweenDuration, 1, 0,
    sizeof(TimeValue), &duration, nil);
```

Once the tween samples have been created, you can add them to the tween media and then add the tween media to the track, as shown in Listing 13-3.

Listing 13-3 Adding the tween sample to the media and the media to the track

```
// add the sample to the tween media
BeginMediaEdits (md);
AddMediaSample (md, container, 0,
    GetHandleSize(container), kSampleDuration, desc, 1, 0, nil);
EndMediaEdits(md);

// dispose of the sample description handle and the atom container
DisposeHandle ((Handle)desc);
QTDisposeAtomContainer(container);

// add the media to the track
InsertMediaIntoTrack(t, 0, 0, kSampleDuration, kFix1);
```

Once you have added the tween media to its track, you need to call the AddTrackReference function to create a link between the tween track to the receiving track. AddTrackReference returns the index of the reference it creates.

The sample code shown in Listing 13-4 retrieves the sound track from a movie and calls AddTrackReference to create a link between the tween track (t) and the sound track. The reference index is returned in the parameter referenceIndex.

Listing 13-4 Creating a link between the tween track and the sound track

```
Track soundTrack;
long referenceIndex;

// retrieve the sound track from the movie
soundTrack = GetMovieIndTrackType (theMovie, 1,
    AudioMediaCharacteristic,
    movieTrackCharacteristic | movieTrackEnabledOnly);

// create a link between the tween track and the sound track --
// on return, referenceIndex contains the index of the link
err = AddTrackReference (soundTrack, t, kTrackModifierReference,
                            &referenceIndex);
```

Once you have linked the tween track to its receiving track, you must update the input map of the receiving track's media to indicate how the receiving track should interpret the data it receives from the tween track. To do this, you create a QT atom container and insert an atom of type `kTrackModifierInput` whose ID is the index returned by the `AddTrackReference` function. Then, you insert two atoms as children of the `kTrackModifierInput` atom:

- A leaf atom of type `kTrackModifierType` that contains the attribute of the receiving track to be modified. For example, if the tween entry modifies the matrix of the track, the leaf atom would contain the type `kTrackModifierTypeMatrix`.
- A leaf atom of type `kInputMapSubInputID` that contains the ID of the tween entry atom. This binds the tween entry to the receiving track.

Once you have created the appropriate atoms in the input map, you call `SetMediaInputMap` to assign the input map to the receiving track's media.

The code shown in Listing 13-5 creates an input map for the sound track of a movie. In this code, the tween media is linked to a sound track; the interpolated tween values are used to modify the sound track's volume.

Listing 13-5 Binding a tween entry to its receiving track

```
QTAtomContainer inputMap = nil;

// create an atom container to hold the input map
if (QTNewAtomContainer (&inputMap) == noErr)
{
    QTAtom inputAtom;
    OSType inputType;
    long tweenID = 1;

    // create a kTrackModifierInput atom
    // whose ID is referenceIndex
    QTInsertChild(inputMap, kParentAtomIsContainer,
        kTrackModifierInput, referenceIndex, 0, 0, nil,
        &inputAtom);

    // add a child atom of type kTrackModifierTypeVolume
    inputType = EndianU32_NtoB(kTrackModifierTypeVolume);
    QTInsertChild (inputMap, inputAtom, kTrackModifierType, 1, 0,
       sizeof(inputType), &inputType, nil);

    // add a child atom for the ID of the tween to
    // modify the volume
    QTInsertChild (inputMap, inputAtom, kInputMapSubInputID, 1,
        0, sizeof(tweenID), &tweenID, nil);

    // assign the input map to the sound media
    SetMediaInputMap(GetTrackMedia(soundTrack), inputMap);

    // dispose of the input map
    QTDisposeAtomContainer(inputMap);
}
```

Constants

The following input type is defined for tween-related atoms:

```
enum {
    kInputMapSubInputID         = 'subi',
};
```

The `kInputMapSubInputID` type is the QT atom type for mapping a tween to a receiving track in a movie:

`kInputMapSubInputID`
: A leaf atom that contains the ID of a tween entry. You create a `kInputMapSubInputID` atom in a receiving track's input map to define the relationship between the tween entry and the receiving track.You create a `kInputMapSubInputID` atom as a child of a `kTrackModifierInput` atom.

C H A P T E R 1 4

Preview Components

This chapter discusses new features and changes to preview components as documented in Chapter 12 of *Inside Macintosh: QuickTime Components*.

New Features of Preview Components

Single Fork Preview Support

QuickTime has always supported the display of file previews using the `StandardGetFilePreview` function. The format for storing these file previews has always been based on Macintosh resources. However, this approach does not work for files created or viewed on operating systems that do not support resource forks. You can now use the `StandardGetFilePreview` function to display previews that are stored in the data fork of a file. QuickTime does not, however, provide support for creating previews that are stored in the data fork of a file. Applications must create these previews themselves.

Preview Components Reference

This section describes the new structure associated with preview components. This new structure makes it possible for your application to use `StandardGetFilePreview` to display previews stored entirely in the data fork of a file. Likewise, your application can create file previews and store them in the data fork of a file so they can be viewed by users of other operating systems.

Resources

The Preview Resource

If your application creates previews, you may want to write them using the data fork format so they can be used on any platform on which QuickTime is available.

The preview display code assumes that the data fork of the file is formatted using QuickTime atoms. See *QuickTime File Format Specification, May 1996* for information on atom-based storage.

Adding a preview results in at least two atoms being added to the data file. The first atom has a `pnot` tag. Its basic structure is the same as the `pnotResource` structure.

```
struct PreviewResourceRecord {
    unsigned long       modDate;
    short               version;
    OSType              resType;
    short               resID;
};
```

Field descriptions

`modDate`	Contains the modification time (in the standard Macintosh format of seconds since midnight, January 1, 1904) of the file for which the preview was created. This parameter allows you to find out if the preview is out of date with the contents of the file.
`version`	Contains the version number of the preview resource. The low bit of the version is a flag for preview components that only reference their data. If the bit is set, it indicates that the resource identified in the preview resource is not owned by the preview component, but is part of the file. It is not removed when the preview is updated or removed (using the Image Compression Manager's `MakeFilePreview` or `AddFilePreview` function), as it would if the version number were 0.

`resType`	Identifies the type of the preview component used to display the preview data and the type of the atom containing the preview data.
`resID`	Contains the index (1-based) of the atom to be used. For example, a `resType` of PICT and a `resID` of 2 tells QuickTime to use the second PICT atom in the file for the preview data.

C H A P T E R 1 5

Data Handler Components

This chapter describes data handler components. **Data handler components** allow QuickTime to retrieve time-based data from external storage devices and, in some cases, store time-based data on those devices.

In most cases, you do not need to create a data handler or use one directly, because QuickTime takes care of data storage and retrieval for you through its built-in media handlers. However, you may need to create a data handler component to read or write to a non-Macintosh storage medium.

Data handler components rely on the facilities of the Component Manager. In order to create or use any component, your application must also use the Component Manager. If you are not familiar with the Component Manager, see Chapter 6 of *Inside Macintosh: More Macintosh Toolbox*. For information about using the Component Manager with QuickTime for Windows, see *Creating Custom Components: QuickTime for Windows*. In addition, you should be familiar with Chapter 1, “Movie Toolbox” in this Reference guide and in *Inside Macintosh: QuickTime* .

About Data Handler Components

A data handler component stores and retrieves time-based data on a storage device, such as a movie file, on behalf of another QuickTime component, typically a media handler component or a sequence grabber component. Different QuickTime components are used depending on if you are retrieving or storing data.

Movie Playback

During data retrieval, such as playback of a movie, a media handler component isolates your application and the Movie Toolbox from the details of how to retrieve data from a particular storage medium. Therefore, unless you are writing your own media handler, you do not have to directly use data handler components in your application; the retrieval of your data will be taken care of for you by the media handler the Movie Toolbox calls. However, you can call the data handler directly if you need to explicitly tell the data handler something, such as to use less memory when caching QuickTime data. If you are reading from a non-Macintosh storage medium, or multiple storage media, you might need to write your own data handler.

Movie Capture

During data storage, such as the capture of video and sound into a movie file, a a sequence grabber component isolates your application from the details of how to capture the raw data from a particular device. Therefore, during movie capture you do not have to directly use data handler components in your application, the storage of your data will be taken care of for you by the sequence grabber component you call. If, however, you are storing data onto a non-Macintosh or proprietary storage medium, or multiple storage media, you might need to write your own data handler.

The sequence grabber component calls the appropriate channel component, such as a video, sound, or text channel component, to retrieve the raw data from an input device, such as a microphone.

Processing data

Data handlers do not know anything about the content of the data they process. It is the responsibility of the client (that is, a media handler component or a channel component) to process the data. In the case of a movie’s video data during movie playback, for example, the media handler takes the data from a data handler and uses the facilities of the Image Compression Manager to display the movie data on the computer screen. See *Inside Macintosh: QuickTime Components* for more information about media handlers.

While data handlers do not work with the content of the data they process, they must be aware of all of the details involved in storing and retrieving data from the storage medium they support. Apple provides several data handlers and a selection mechanism for choosing an appropriate handler. For example, one supports data access from HFS volumes and another supports the memory-based data handler, which allows QuickTime to retrieve movies from memory handles. These two data handler components use very different mechanisms to store and retrieve movie data.

You might need to write your own data handler when you are accessing a storage medium for which there is no Apple-supplied data handler or when playing movies from a multimedia server, as you will need to use a data handler that understands the network protocols and data formats necessary to communicate with that server.

Identifying Containers With Data References

A **container** is the system element that contains the movie data and can be any element that can contain data. For example, a container may be an in-memory data structure, a local disk file, or a file on a networked multimedia server. As is the case throughout QuickTime, all data handlers identify their movie-data containers with data references. Data references identify the location of the container and its type.

Different container types may require different types of references. For example, files are identified using aliases, while memory-based movies are identified by handles. The data reference data type is flexible enough to accommodate all these cases. The data handler component must specify the type of reference it requires and verify that the references supplied by client applications are valid. Data handler components use the component subtype value to specify the reference type they support.

Whenever an application opens a container, the Movie Toolbox determines the most appropriate data handler component to use in order to access that container. The Movie Toolbox makes this determination by querying the various data handlers installed on the user's computer. If your application uses the Movie Toolbox, this selection process is transparent to your program. If you develop your own data handler, your component must support the selection functions, see "Data Handler Components Reference" (page 447), for more information).

Using Data Handler Components

This section describes how applications use data handler components. You should read this section if you are writing your own media handler or your own data handler.

Selecting a Data Handler

To help developers choose the best data handler for a specific situation while still making it easy for an application to find a usable data handler, Apple has defined two separate and complementary mechanisms for selecting data handler components. You can use the Component Manager's selection mechanisms to find a data handler that meets your needs and you can interrogate a data handler to determine if it supports a specific data reference. Both mechanisms rely on characteristics of the current data reference in order to make the selection.

Before you can use a data handler component, your application must open a connection to that component. The easiest way to open a connection to a data handler component is to call the Movie Toolbox's `GetDataHandler` function. You supply a data reference and the Movie Toolbox selects an appropriate data handler component for you. This function is preferred for opening a data handler as it reliably chooses the best data handler. For more information about this function, see the chapter "Movie Toolbox" in *Inside Macintosh: QuickTime*.

Alternatively, you may use the Component Manager to open your connection. Call the Component Manager's `OpenDefaultComponent` or `OpenComponent` function to do so, but be aware that these functions are often unable to make the best choice when there are several different data handlers available for a file.

Selecting by Component Type Value

At the most basic level, your application can use the Component Manager's built-in selection mechanisms to find a data handler component for a data reference. You may use the Component Manager's `FindNextComponent` function in order to retrieve a list of all data handler components that meet your needs. You specify your request by supplying the component's characteristics in a component description record—in particular, in the `componentType`, `componentSubtype`, `componentManufacturer`, and `componentFlags` fields.

All data handler components have a component type value of `'dhlr'`, which is defined by the `dataHandlerType` constant. Data handler components use the value of the component subtype field to indicate the type of data reference they support. As a result of this convention, note that all data handlers that share a component subtype value must be able to recognize and work with data references of the same type. For example, file system data handlers always carry a component subtype value of `'alis'`, which indicates that their data references are file system aliases (note that this is true for QuickTime on the Macintosh and under Windows, even though there is not, properly, a file system alias under Windows). Apple's memory-based data handler for the Macintosh has a component subtype value of `'hndl'`.

Apple has not defined any special manufacturer field values or component flags values for data handler components. You may use the manufacturer field to select data handlers supplied by a specific vendor. To do so, you need to determine the appropriate manufacturer field value for that vendor.

Interrogating a Data Handler's Capabilities

While you can use the Component Manager's selection mechanisms to find a data handler component that can recognize data references of a specific type, your application must interact with the data handler in order to determine whether it can support a specific data reference. Apple has defined two functions, `DataHCanUseDataRef` and `DataHGetVolumeList`, that allow you to query a data handler component in order to find out whether it can work with a data reference. By using these two functions, your application can choose a data handler that is best-suited to its specific needs.

Using the `DataHCanUseDataRef` function, you supply a data reference to the data handler component. The component then reports what it can do with that data reference. The returned value indicates the level and, to some extent, the quality of service the data handler can provide (for example, whether the component can read data from or write data to the data reference and whether the component uses any special support when working with that data reference).

Because calling the `DataHCanUseDataRef` function in several data handlers can get time consuming, Apple has also defined a function that helps narrow the search. By using the `DataHGetVolumeList` function, your application can obtain a list of all the file system volumes that a data handler can support. In response to your request, the data handler returns the list and flags indicating the level and quality of service the data handler can provide for containers on that volume.

For more information on these functions, see “Selecting a Data Handler” (page 447).

Managing Data References

Once you have selected a data handler component, you must provide a data reference to the data handler. Use the `DataHSetDataRef` function to supply a data reference to a data handler. Once you have assigned a data reference to the data handler, your application may start reading and/or writing movie data from that data reference. The `DataHGetDataRef` function allows your application to obtain a data handler’s current data reference.

Data handlers also provide a function that allows your application to determine whether two data references are equivalent (that is, refer to the same movie container). Your application provides a data reference to the `DataHCompareDataRef` function. The data handler returns a Boolean value indicating whether that data reference matches the data handler’s current data reference.

For more information on these functions, see “Working With Data References” (page 454).

Retrieving Movie Data

Before your application can read data using a data handler component, you must open a read path to the current data reference. Use the `DataHOpenForRead` function to request read access to the current data reference. Once you have gained read access to the data reference, data handlers provide both high- and low-level read functions.

The high-level function, `DataHGetData`, provides an easy-to-use, synchronous read interface. Being a synchronous function, `DataHGetData` does not return control to your application until the data handler has read and delivered the data you request.

If you need more control over the read operation, you can use the low-level function, `DataHScheduleData`, to issue asynchronous read requests. When you call this function, you provide detailed information specifying when you need the data from the request. The data handler returns control to your application immediately, and then processes the request when appropriate. When the data handler completes the request, it calls your data-handler completion function to report that the request has been satisfied, see “Completion Function” (page 479) for more information on the data-handler completion function.

Besides simply scheduling read requests that must be satisfied during a movie’s playback, another use of the `DataHScheduleData` function is to prepare a movie for playback (commonly referred to as pre-rolling the movie). The `DataHScheduleData` function uses several special values to indicate a pre-roll operation. Your application calls the `DataHScheduleData` function one or more times to schedule the pre-roll read requests, and then uses the `DataHFinishData` function to tell the data handler to start delivering the requested data.

For more information on these functions and about pre-roll operations, see “Reading Movie Data” (page 459).

Storing Movie Data

Before your application can write data using a data handler component, you must open a write path to the current data reference. Use the `DataHOpenForWrite` function to request write access to the current data reference. Once you have gained write access to the data reference, data handler components provide both high- and low-level write functions.

Note
QuickTime for Windows version 2.1.1 or earlier does not support writing movie data. ◆

The high-level function, `DataHPutData`, allows you to easily append data to the end of the container identified by a data reference. Except when capturing movie data using the sequence grabber component, the Movie Toolbox uses this call when writing data to movie files. However, this function does not allow your application to write to any location other than the end of the container. In addition, this is a synchronous operation, so control is not returned to your program until the write is complete. As a result, this function is not well-suited to high-performance write operations, such as would be required to capture a movie.

If you need a more flexible write facility, or one with higher performance characteristics, you can use the `DataHWrite` function. This function is intended to support high-speed writes, suitable for movie capture operations. For example, Apple's sequence grabber component uses this data handler function to capture movies.

When you call this function, you provide detailed information specifying the location in the container that is to receive the data. The data handler returns control to your application immediately, and then processes the request asynchronously. When the data handler completes the request, it calls your data-handler completion function to report that the request has been satisfied, see "Completion Function" (page 479) for more information on the data-handler completion function.

In addition to the `DataHWrite` function, data handler components provide several other "helper" functions that allow you to create new movie containers and prepare them for a movie capture operation.

For more information on all of these functions, see "Writing Movie Data" (page 468).

Managing the Data Handler

Data handler components provide a number of functions that your application can use to manage its connection to the handler. The most important among these is `DataHTask`, which provides processor time to the handler. Your application should call this function often so that the handler has enough time to do its work.

Other functions in this category provide playback hints to the data handler and allow your application to influence how the component handles its cached data.

For more information on these functions, see "Managing Data Handler Components" (page 476).

Creating a Data Handler Component

This section describes the details of creating a data handler component and includes source code for a simple data handler component. After reading this section, you will understand all of the special requirements of these components. The functional interface that your component must support is described in "Data Handler Components Reference" (page 447).

You should consider developing your own data handler component if you are planning to provide a new type of movie container or a container that requires special data handling techniques. For example, if you are planning to develop a networked multimedia server, you would most likely need to develop a new data handler that could support the special protocols required by your server. By encapsulating that protocol support in a data handler, QuickTime applications can access the movie data on your server without having to implement any special support. In this way, your server becomes a seamless part of the user's system.

Before reading this section, you should be familiar with how to create components. See "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for a complete description of components, how to use them, and how to create them on the Macintosh. For further information about using the Component Manager with QuickTime for Windows, see *Creating Custom Components: QuickTime for Windows.*

General Information

All data handler components have a component type value of `'dhlr'`, which is defined by the `dataHandlerType` constant. Data handler components use the value of the component subtype field to indicate the type of data reference they support. As a result of this convention, note that all data handlers that share a component subtype value must be able to recognize and work with data references of the same type. For example, file system data handlers always carry a component subtype value of `'alis'`, which indicates that their data references are file system aliases (note that this is true for QuickTime on the Macintosh and under Windows, even though there is not, properly, a file system alias under Windows). Apple's memory-based data handler for the Macintosh has a component subtype value of `'hndl'`.

```
#define dataHandlerType 'dhlr'
#define rAliasType      'alis'
```

Apple has not defined any special manufacturer field values or component flags values for data handler components. Developers may use the manufacturer field value to select your data handler from among all the data handlers that support a given type of data reference.

Apple has defined a functional interface for data handler components. For information about the functions that your component must support, see "Data

Handler Components Reference" (page 447). You can use the following constants to refer to the request codes for each of the functions that your component must support:

```
enum {

    kDataGetDataSelector    = 2,    /* DataHGetData */
    kDataPutDataSelector    = 3,    /* DataHPutData */
    kDataFlushDataSelector= 4,  /* DataHFlushData */
    kDataOpenForWriteSelector= 5,/* DataHOpenForWrite */
    kDataCloseForWriteSelector= 6,/* DataHCloseForWrite */
    kDataOpenForReadSelector= 8,/* DataHOpenForRead */
    kDataCloseForReadSelector= 9,/* DataHCloseForRead */
    kDataSetDatRefSelector= 10, /* DataHSetDataRef */
    kDataGetDataRefSelector= 11,/* DataHGetDataRef */
    kDataCompareDataRefSelector= 12,/* DataHCompareDataRef */
    kDataTaskSelector           = 13,   /* DataHTask */
    kDataScheduleDataSelector= 14,/* DataHScheduleData */
    kDataFinishDataSelector= 15,/* DataHFinishData */
    kDataFlushCacheSelector= 16,/* DataHFlushCache */
    kDataResolveDataRefSelector= 17,/* DataHResolveDataRef
*/
    kDataGetFileSizeSelector= 18,/* DataHGetFileSize */
    kDataCanUseDataRefSelector= 19,/* DataHCanUseDataRef */
    kDataGetVoumeListSelector= 20,/* DataHGetVolumeList */
    kDataWriteSelector      = 21,   /* DataHWrite */
    kDataPreextendSelector= 22, /* DataHPreextend */
    kDataSetFileSizeSelector= 23,/* DataHSetFileSize */
    kDataGetFreeSpaceSelector= 24,/* DataHGetFreeSpace */
    kDataCreateFileSelector= 25,/* DataHCreateFile */
    kDataGetPreferredBlockSizeSelector= 26,/* DataHGetPreferredBlockSize
*/
    kDataGetDeviceIndexSelector= 27,/* DataHGetDeviceIndex */
    /* 28 and 29 unused */
    kDataGetScheduleAheadTimeSelector= 30,/*

DataHGetScheduleAheadTime */
    kDataSetOSFileRefSelector= 516,/* DataHSetOSFileRef */
    kDataGetOSFileRefSelector= 517,/* DataHGetOSFileRef */
    kDataPlaybackHintsSelector= 3+0x100/* DataHPlaybackHints */
};
```

Data Handler Components Reference

This section describes the functions your data handler component may support. Some of these functions are optional—your component should support only those functions that are appropriate to it.

Data Handler Components Functions

This section describes the functions that may be supported by data handler components.

Selecting a Data Handler

In order for client programs to choose the best data handler component for a data reference, Apple has defined some functions that allow applications to interrogate a data handler's capabilities.

The `DataHGetVolumeList` function allows an application to obtain a list of the volumes your data handler can support. The `DataHCanUseDataRef` function allows your data handler to examine a specific data reference and indicate its ability to work with the associated container. The `DataHGetDeviceIndex` function allows applications to determine whether different data references identify containers that reside on the same device.

By way of illustration, the Movie Toolbox uses the `DataHGetVolumeList` and `DataHCanUseDataRef` functions as follows. During startup, and whenever a new volume is mounted, the Movie Toolbox calls each data handler's `DataHGetVolumeList` function in order to obtain information about each handler's general capabilities. Specifically, the Movie Toolbox calls each component's `GetDataHandler`, `DataHGetVolumeList`, and `CloseComponent` functions.

Whenever an application opens a movie, the Movie Toolbox selects the best data handler for the movie's container. This may involve calling each appropriate data handler's `DataHCanUseDataRef` function (in some cases, a data handler may indicate that it does not need to examine a data reference before accessing it—see the description of the `DataHGetVolumeList` function for more information). For each data handler that can support the data reference (that is, has the correct component subtype value) and needs to be interrogated, the Movie Toolbox calls the component's `GetDataHandler`, `DataHCanUseDataRef`, and `CloseComponent` functions. Based on the resulting information, the Movie Toolbox selects the best data handler for the application.

For more information on selecting a data handler, see "Selecting a Data Handler" (page 440).

DataHGetVolumeList

In response to the `DataHGetVolumeList` function, your data handler component returns a list of the volumes your component can access, along with flags indicating your component's capabilities for each volume.

```
pascal ComponentResult DataHGetVolumeList (DataHandler dh,
                          DataHVolumeList *volumeList);
```

`dh`	Identifies the calling program's connection to your data handler component.
`volumeList`	Contains a pointer to a field that your data handler component uses to return a handle to a volume list. Your component constructs the volume list by allocating a handle and filling it with a series of `DataHVolumeListRecord` structures (one structure for each volume your component can access). This structure is described later in this section.

DISCUSSION

In order to reduce the delay that may result from choosing an appropriate data handler for a volume, the Movie Toolbox maintains a list of data handlers and the volumes they support. The Movie Toolbox uses the `DataHGetVolumeList` function to build that list.

When your component receives this function, it should scan the available volumes and create a series of `DataHVolumeListRecord` structures—one structure for each volume your component can access. This structure is defined as follows:

```
typedef struct DataHVolumeListRecord {
short        vRefNum; /* reference number */
long         flags;  /* capability flags */
} DataHVolumeListRecord, *DataHVolumeListPtr, **DataHVolumeList;
```

`vRefNum`	Contains the volume reference number assigned to the volume.
`flags`	Indicates the level of support your data handler can provide for this volume. These flags are similar to those defined for the `DataHCanUseDataRef` function, though there is one additional flag. Your component should set every appropriate flag to 1 (set unused flags to 0).

`kDataHCanRead`
: Indicates that your data handler can read from the volume.

`kDataHSpecialRead`
: Indicates that your data handler can read from the volume using a specialized method. For example, your data handler might support access to networked multimedia servers using a special protocol. In that case, your component would set this flag to 1 whenever the volume resides on a supported server.

`kDataHSpecialReadFile`
: Reserved for use by Apple.

`kDataHCanWrite`
: Indicates that your data handler can write data to the volume. In particular, use this flag to indicate that your data handler's `DataHPutData` function will work with this volume.

`kDataHSpecialWrite`
: Indicates that your data handler can write to the volume using a specialized method. As with the `kDataHSpecialRead` flag, your data handler would

use this flag to indicate that your component can access the volume using specialized support (for example, special network protocols).

`kDataHCanStreamingWrite`
: Indicates that your data handler can support the special write functions for capturing movie data when writing to this volume. These functions are described in“Writing Movie Data” (page 468) “ of this chapter.

`kDataHMustCheckDataRef`
: Instructs the calling program that your component must check each data reference before it can accurately report its capabilities. If you set this flag to 1, the Movie Toolbox will call your component’s `DataHCanUseDataRef` function before it assigns a container to your data handler. Note, however, that this may slow the data handler selection process somewhat.

Your data handler may use any facilities necessary to determine whether it can access the volume, including opening a container on the volume. Your component should set to 1 as many of the capability flags as are appropriate for each volume. Do not include records for volumes your handler cannot support.

For example, if your component supports networked multimedia servers using a special set of protocols, your data handler should set the `kDataHCanRead` and `kDataHCanSpecialRead` flags to 1 for any volume that is on that server. In addition, if your component can write to a volume on the server, set the `kDataHCanWrite` and `kDataHCanSpecialWrite` flags to 1 (perhaps along with `kDataHCanStreamingWrite`). However, your component should create entries only for those volumes that support your protocols.

It is the calling program’s responsibility to dispose of the handle returned by your component.

The Movie Toolbox tracks mounting and unmounting removable volumes, and keeps its volume list current. As a result, the Movie Toolbox may call your component’s `DataHGetVolumeList` function whenever a removable volume is mounted.

If your data handler does not process data that is stored in file system volumes, you need not support this function.

RESULT CODEs

Memory Manager errors

DataHCanUseDataRef

The `DataHCanUseDataRef` function allows your data handler to report whether it can access the data associated with a specified data reference.

```
pascal ComponentResult DataHCanUseDataRef (DataHandler dh, Handle
                        dataRef, long *useFlags);
```

`dh`	Identifies the calling program's connection to your data handler component.
`dataRef`	Specifies the data reference. This parameter contains a handle to the information that identifies the container in question.
`useFlags`	Contains a pointer to a field that your data handler component uses to indicate its ability to access the container identified by the `dataRef` parameter. Your data handler may use the following flags (set all flags that are appropriate to 1; set unused flags to 0):

`kDataHCanRead`
: Indicates that your data handler can read from the container.

`kDataHSpecialRead`
: Indicates that your data handler can read from the container using a specialized method. For example, your data handler might support access to networked multimedia servers using a special protocol. In that case, your component would set this flag to 1 whenever the data reference identifies a container on a supported server.

`kDataHSpecialReadFile`
: Indicates that your data handler can read from the container using a specialized method that is particular to the type of container in question.

For example, your data handler may use a different method for some types of containers (say, a Hypercard stack).

This flag represents a special case of the `kDataHSpecialRead` flag. That is, this flag is appropriate only if you have also set `kDataHSpecialRead` to 1.

`kDataHCanWrite`
: Indicates that your data handler can write data to the container. In particular, use this flag to indicate that your data handler's `DataHPutData` function will work with this data reference.

`kDataHSpecialWrite`
: Indicates that your data handler can write to the container using a specialized method. As with the `kDataHSpecialRead` flag, your data handler would use this flag to indicate that the data reference identifies a container which your component can access using specialized support (for example, special network protocols).

`kDataHCanStreamingWrite`
: Indicates that your data handler can support the special write functions for capturing movie data when writing to this container. These functions are described later in this chapter, in "Writing Movie Data."

If your data handler cannot access the container, set the field to 0.

DISCUSSION

Apple's standard data handler sets both the `kDataHCanRead` and `kDataHCanWrite` flags to 1 for any data reference it receives, indicating that it can read from and write to any volume.

Your component should set to 1 as many of the capability flags as are appropriate for the specified data reference. Conversely, be sure to set the flags to 0 if your component cannot support the container. For example, if your component supports networked multimedia servers using a special set of protocols, your data handler should set the `kDataHCanRead` and `kDataHCanSpecialRead` flags to 1 for any container that is on that server. In addition, if your component can write to the server, set the `kDataHCanWrite` and `kDataHCanSpecialWrite` flags to 1 (perhaps along with `kDataHCanStreamingWrite`). However, your component should set the flags field to 0 for any container that is not on a server that supports your protocols.

Your data handler may use any facilities necessary to determine whether it can access the container. Bear in mind, though, that your component should try to be as quick about this determination as possible, in order to minimize the chance that the delay will be noticed by the user.

SEE ALSO

The Movie Toolbox calls your component's `DataHGetVolumeList` function to retrieve your data handler's capabilities for an entire volume.

DataHGetDeviceIndex

In response to the `DataHGetDeviceIndex` function, your data handler component returns a value that identifies the device on which a data reference resides.

```
pascal ComponentResult DataHGetDeviceIndex (DataHandler dh, long
                       *deviceIndex);
```

`dh`	Identifies the calling program's connection to your data handler component.
`deviceIndex`	Contains a pointer to a field that your data handler component uses to return a device identifier value.

DISCUSSION

Some client programs may need to account for the fact that two or more data references reside on the same device. For instance, this may affect storage-allocation requirements. This function allows such client programs to obtain this information from your data handler.

Your component may use any identifier value that is appropriate (as an example, Apple's HFS data handler uses the volume reference number). The client program should do nothing with the value other than compare it with other identifiers returned by your data handler component.

Working With Data References

All data handler components use data references to identify and locate a movie's container. Different types of containers may require different types of data references. For example, a reference to a memory-based movie may be a handle, while a reference to a file-based movie may be an alias.

Client programs can correlate data references with data handlers by matching the component's subtype value with the data reference type—the subtype value indicates the type of data reference the component supports. All data handlers with the same subtype value must support the same data reference type. To continue the previous example, Apple's memory-based data handler for the Macintosh uses handles (and has a subtype value of `'hndl'`), while the HFS data handler uses Alias Manager aliases (its subtype value is `'alis'`).

The `DataHSetDataRef` and `DataHGetDataRef` functions allow applications to assign your data handler's current data reference. The `DataHCompareDataRef` function asks your component to compare a data reference against the current data reference and indicate whether the references are equivalent (that is, refer to the same container). The `DataHResolveDataRef` permits your component to locate a data reference's container.

The `DataHSetOSFileRef` and `DataHGetOSFileRef` functions provide an alternative, system-specific mechanism for assigning your data handler's current data reference.

For more information on data references, see "Managing Data References" (page 442).

DataHSetDataRef

The `DataHSetDataRef` function assigns a data reference to your data handler component.

```
pascal ComponentResult DataHSetDataRef (DataHandler dh, Handle dataRef);
```

`dh`	Identifies the calling program's connection to your data handler component.
`dataRef`	Specifies the data reference. This parameter contains a handle to the information that identifies the container in question. Your component must make a copy of this handle.

DISCUSSION

Note that the type of data reference always corresponds to the type that your component supports, and that you specify in the component subtype value of your data handler. As a result, the client program does not provide a data reference type value (unlike the Movie Toolbox's data reference functions).

The client program is responsible for disposing of the handle. Consequently, your component must make a copy of the data reference handle.

RESULT CODEs

Memory Manager errors

DataHGetDataRef

The `DataHGetDataRef` function retrieves your component's current data reference.

```
pascal ComponentResult DataHGetDataRef (DataHandler dh, Handle *dataRef);
```

`dh`	Identifies the calling program's connection to your data handler component.
`dataRef`	Contains a pointer to a data reference handle. Your component should make a copy of its current data reference in a handle and return that handle in this field. The client program is responsible for disposing of that handle.

RESULT CODEs

Memory Manager errors

DataHCompareDataRef

Your component compares a supplied data reference against its current data reference and returns a Boolean value indicating whether the data references are equivalent (that is, the two data references identify the same container).

```
pascal ComponentResult DataHCompareDataRef (DataHandler dh, Handle
                        dataRef, Boolean *equal);
```

`dh`	Identifies the calling program's connection to your data handler component.
`dataRef`	Specifies the data reference to be compared to your component's current data reference.
`equal`	Contains a pointer to a Boolean. Your component should set that Boolean to `true` if the two data references identify the same container. Otherwise, set the Boolean to `false`.

DISCUSSION

Note that your component cannot simply compare the bits in the two data references. For example, two completely different aliases may refer to the same HFS file. Consequently, you need to completely resolve the data reference in order to determine the file identified by the reference.

DataHResolveDataRef

The `DataHResolveDataRef` function instructs your data handler component to locate the container associated with a given data reference.

```
pascal ComponentResult DataHResolveDataRef (DataHandler dh, Handle
                        theDataRef, Boolean *wasChanged,
                        Boolean userInterfaceAllowed);
```

`dh`	Identifies the calling program's connection to your data handler component.
`theDataRef`	Specifies the data reference to be resolved.

`wasChanged` Contains a pointer to a Boolean. Your component should set that Boolean to `true` if, in locating the container, your data handler updates any information in the data reference.

`userInterfaceAllowed`
Indicates whether your component may interact with the user when locating the container. If this parameter is set to `true`, your component may ask the user to help locate the container (for instance, by presenting a Find File dialog box).

DISCUSSION

This function is, essentially, equivalent to the Alias Manager's `ResolveAlias` function. The client program asks your component to locate the container that is associated with a given data reference. If your component determines that the data reference needs to be updated with more accurate location information, it should put the new information in the supplied data reference (and set the Boolean referred to by the `wasChanged` parameter to `true`).

Client programs may call your data handler's `DataHResolveDataRef` function at any time. Typically, however, the Movie Toolbox uses this function as part of its strategy for opening and reading a movie container. As such, you can expect that the supplied data reference will identify a container that your component can support.

DataHSetOSFileRef

The `DataHSetOSFileRef` function assigns a movie container to your data handler component. Applications may use this function instead of calling the `DataHSetDataRef` function in cases where the applications have already opened the container.

```
pascal ComponentResult DataHSetOSFileRef (DataHandler dh, long ref, long
                        flags);
```

`dh` Identifies the calling program's connection to your data handler component.

`ref`	Specifies the container. This parameter contains an operating system-specific file-access token. For example, on the Macintosh an application would supply the file reference it obtained by calling the `FSOpenFile` function. Under Windows, this parameter would contain an `HFILE` value obtained from the `OpenFile` function.
`flags`	Specifies access flags for the container. This parameter contains the access flags the application used when opening the container. Again, these are operating system-specific.

DISCUSSION

This function provides an alternative mechanism for assigning your data handler's current container. In some cases, an application may have created or opened a movie container prior to assigning the container to your handler. In such cases, the application may choose to provide its access token to your data handler, rather than using the `DataHSetDataRef` function to assign a data reference. The application must have opened the file before calling this function.

Note that your data handler must implement this function in a system-specific manner, and must verify that the access token is valid.

Applications must still call your handlers `DataHOpenForRead` or `DataHOpenForWrite` functions, as appropriate, before using your data handler to access the container.

RESULT CODEs

`invalidDataRef`	–2012	Application already set a data reference
`memFullErr`	–108	Insufficient memory for operation

DataHGetOSFileRef

The `DataHGetOSFileRef` function retrieves your component's container access token, if it was assigned using the `DataHSetOSFileRef` function.

```
pascal ComponentResult DataHGetOSFileRef (DataHandler dh, long *ref, long
                        *flags);
```

`dh`	Identifies the calling program's connection to your data handler component.
`ref`	Contains a pointer to a long. Your component should return the container access token that the application provided when it called your `DataHSetOSFileRef` function.
`flags`	Contains a pointer to a long. Your component should return the access flags that the application provided when it called your `DataHSetOSFileRef` function.

RESULT CODEs

`invalidDataRef`	–2012	Application already set a data reference
`memFullErr`	–108	Insufficient memory for operation

Reading Movie Data

Data handler components provide two basic read facilities. The `DataHGetData` function is a fully synchronous read operation, while the `DataHScheduleData` function is asynchronous. Applications provide scheduling information when they call your component's `DataHScheduleData` function. When your component processes the queued request, it calls the application's data-handler completion function (for more information, see "Completion Function" (page 479) later in this chapter). By calling your component's `DataHFinishData` function, applications can force your component to process queued read requests. Applications may call your component's `DataHGetScheduleAheadTime` function in order to determine how far in advance your component prefers to get read requests.

Before any application can read data from a data reference, it must open read access to that reference by calling your component's `DataHOpenForRead` function. The `DataHCloseForRead` function closes that read access path.

For more information on reading movie data, see "Retrieving Movie Data" (page 442).

DataHOpenForRead

Your component opens its current data reference for read-only access.

```
pascal ComponentResult DataHOpenForRead (DataHandler dh);
```

`dh`	Identifies the calling program's connection to your data handler component.

DISCUSSION

After setting your component's current data reference by calling the `DataHSetDataRef` function, client programs call the `DataHOpenForRead` function in order to start reading from the data reference. Your component should open the data reference for read-only access. If the data reference is already open or cannot be opened, return an appropriate error code.

Note that the Movie Toolbox may try to read data from a data reference without calling your component's `DataHOpenForRead` function. If this happens, your component should open the data reference for read-only access, respond to the read request, and then leave the data reference open in anticipation of later read requests.

DataHCloseForRead

Your component closes read-only access to its data reference.

```
pascal ComponentResult DataHCloseForRead (DataHandler dh);
```

`dh`	Identifies the calling program's connection to your data handler component.

DISCUSSION

Note that a client program may close its connection to your component (by calling the Component Manager's `CloseComponent` function) without closing the read path. If this happens, your component should close the data reference before closing the connection.

RESULT CODES

`dataNotOpenForRead`	–2042	Data reference not open for read
`dataAlreadyClosed`	–2045	This reference already closed

DataHGetData

Your component reads data from its current data reference. This is a synchronous read operation.

```
pascal ComponentResult DataHGetData (DataHandler dh, Handle h, long
                        hOffset, long offset, long size);
```

`dh`	Identifies the calling program's connection to your data handler component.
`h`	Specifies the handle to receive the data.
`hOffset`	Identifies the offset into the handle where your component should return the data.
`offset`	Specifies the offset in the data reference from which your component is to read.
`size`	Specifies the number of bytes to read.

DISCUSSION

The `DataHGetData` function provides a high-level read interface. This is a synchronous read operation; that is, the client program's execution is blocked until your component returns control from this function. As a result, most time-critical clients use the `DataHScheduleData` function to read data.

Note that the Movie Toolbox may try to read data from a data reference without calling your component's `DataHOpenForRead` function. If this happens, your component should open the data reference for read-only access, respond to the read request, and then leave the data reference open in anticipation of later read requests.

SEE ALSO

Client programs can force your component to invalidate any cached data by calling your component's `DataHFlushCache` function.

DataHScheduleData

Your component reads data from its current data reference. This can be a synchronous read operation or an asynchronous read operation.

```
pascal ComponentResult DataHScheduleData (DataHandler dh,
                        Ptr placeToPutDataPtr, long fileOffset, long
                        dataSize, long refCon, DataHSchedulePtr scheduleRec,
                        DHCompletionUPP completionRtn);
```

`dh`	Identifies the calling program's connection to your data handler component.
`placeToPutDataPtr`	Specifies the location in memory that is to receive the data.
`fileOffset`	Specifies the offset in the data reference from which your component is to read.
`dataSize`	Specifies the number of bytes to read.
`refCon`	Contains a reference constant that your data handler component should provide to the data-handler completion function specified with the `completionRtn` parameter.
`scheduleRec`	Contains a pointer to a schedule record. If this parameter is set to `nil`, then the client program is requesting a synchronous read operation (that is, your data handler must return the data before returning control to the client program). If this parameter is not set to `nil`, it must contain the location of a schedule record that has timing information for an asynchronous read request. Your data handler should return control to the client program immediately, and then call the client's data-handler completion function when the data is ready. The schedule record is described later in this section.

`completionRtn`

Contains a pointer to a data-handler completion function. When your data handler finishes with the client program's read request, your component must call this routine. Be sure to call this routine even if the request fails. Your component should pass the reference constant that the client program provided with the `refCon` parameter.

The client program must provide a completion routine for all asynchronous read requests (that is, all requests that include a valid schedule record). For synchronous requests, client programs should set this parameter to `nil`. However, if the function is provided, your handler must call it, even after synchronous requests.

DISCUSSION

The `DataHScheduleData` function provides both a synchronous and an asynchronous read interface. Synchronous read operations work like the `DataHGetData` function—the data handler component returns control to the client program only after it has serviced the read request. Asynchronous read operations allow client programs to schedule read requests in the context of a specified QuickTime time base. Your data handler queues the request and immediately returns control to the calling program. After your component actually reads the data, it calls the client program's data-handler completion function.

If your component cannot satisfy the request (for example, the request requires data more quickly than you can deliver it), your component should reject the request immediately, rather than queuing the request and then calling the client's data-handler completion function.

The client program provides scheduling information for scheduled reads in a schedule record. This structure is defined as follows:

```
typedef struct DataHScheduleRecord {
                    TimeRecord timeNeededBy; /* schedule info */
                    long extendedID; /* type of data */
                    long extendedVers; /* reserved */
                    Fixedpriority;  /* priority */
                    } DataHScheduleRecord, *DataHSchedulePtr;
```

`timeNeededBy` Specifies the time at which your data handler must deliver the requested data to the calling program. This time value is relative to the time base that is contained in this time record.

During pre-roll operations, the Movie Toolbox may use special values in certain time record fields. The time record fields in question are the `scale` and `value` fields. By correctly interpreting the values of these fields, your data handler can queue up the pre-roll read requests in the most efficient way for its device.

There are two types of pre-roll read operations. The first type is a required read; that is, the Movie Toolbox requires that the read operation be satisfied before the movie starts playing. The second type is an optional read. If your data handler can satisfy the read operation as part of the pre-roll operation, it should do so. Otherwise, your data handler may satisfy the request at a specified time while the movie is playing.

The Movie Toolbox indicates that a pre-roll read request is required by setting the `scale` field of the time record to –1. This literally means that the request is scheduled for a time that is infinitely far into the future. Your data handler should collect all such read requests, order them most efficiently for your device, and process them when the Movie Toolbox calls your component's `DataHFinishData` function.

For optional pre-roll read requests, the Movie Toolbox sets the `scale` field properly, but negates the contents of the `value` field. Your data handler has the option of delivering the data for this request with the required data, if that can be done efficiently. Otherwise, your data handler may deliver the data at its schedule time. You determine the scheduled time by negating the contents of the `value` field (that is, multiplying by –1).

For more information about pre-roll operations, see "Retrieving Movie Data," earlier in this chapter.

`extendedID` Indicates the type of data that follows in the remainder of the record. The following values are valid:

`kDataHExtendedSchedule`
The remainder of the record contains extended scheduling information.

If the `extendedID` field is set to `kDataHExtendedSchedule`, the remainder of the schedule record is defined as follows:

`extendedVers`	Reserved; this field should always be set to 0.
`priority`	Indicates the relative importance of the data request. Client programs assign a value of 100.0 to data requests the must be delivered. Lower values indicate relatively less critical data. If your data handler must accommodate bandwidth limitations when delivering data, your component may use this value as an indication of which requests can be dropped with the least impact on the client program. As an example, consider using priorities in a frame-differenced movie. Key frames might have priority values of 100.0, indicating that they are essential to proper playback. As you move through the frames following a key frame, each successive frame might have a lower priority value. Once you drop a frame, you must drop all successive frames of equal or lower priority until you reach another key frame, because each of these frames would rely on the dropped one for some image data.

Note that the Movie Toolbox may try to read data from a data reference without calling your component's `DataHOpenForRead` function. If this happens, your component should open the data reference for read-only access, respond to the read request, and then leave the data reference open in anticipation of later read requests.

SEE ALSO

Client programs can force your component to invalidate any cached data by calling your component's `DataHFlushCache` function.

DataHFinishData

The `DataHFinishData` function instructs your data handler component to complete or cancel one or more queued read requests. The client program would have issued those read requests by calling your component's `DataHScheduleData` function.

```
pascal ComponentResult DataHFinishData (DataHandler dh,
                          Ptr placeToPutDataPtr, Boolean cancel);
```

`dh`	Identifies the calling program's connection to your data handler component.
`placeToPutDataPtr`	Specifies the location in memory that is to receive the data. The value of this parameter identifies the specific read request to be completed. If this parameter is set to `nil`, the call affects all pending read requests.
`cancel`	Indicates whether the calling program wants to cancel the outstanding request. If this parameter is set to `true`, your data handler should cancel the request (or requests) identified by the `placeToPutDataPtr` parameter.

DISCUSSION

Client programs use the `DataHFinishData` function either to cancel outstanding read requests or to demand that the requests be serviced immediately. Pre-roll operations are a special case of the immediate service request. The client program will have queued one or more read requests with their scheduled time of delivery set infinitely far into the future. Your data handler queues those requests until the client program calls the `DataHFinishData` function demanding that all outstanding read requests be satisfied immediately.

Note that your component must call the client program's data-handler completion function for each queued request, even though the client program called the `DataHFinishData` function. Be sure to call the completion function for both canceled and completed read requests.

SEE ALSO

Client programs queue read requests by calling your component's `DataHScheduleData` function.

DataHGetScheduleAheadTime

The `DataHGetScheduleAheadTime` function allows your data-handler component to report how far in advance it prefers clients to issue read requests.

```
pascal ComponentResult DataHGetScheduleAheadTime (DataHandler dh,
                        long *millisecs);
```

`dh`	Identifies the calling program's connection to your data handler component.
`millisecs`	Contains a pointer to a long. Your component should set this field with a value indicating the number of milliseconds you prefer to receive read requests in advance of the time when the data must be delivered.

DISCUSSION

This function allows your data handler to tell the client program how far in advance it should schedule its read requests. By default, the Movie Toolbox issues scheduled read requests between 1 and 2 seconds before it needs the data from those requests. For some data handlers, however, this may not be enough time. For example, some data handlers may have to accommodate network delays when processing read requests. Client programs that call this function may try to respect your component's preference.

Note, however, that not all client programs will call this function. Further, some clients may not be able to accommodate your preferred time in all cases, even if they have asked for your component's preference. As a result, your component should have a strategy for handling requests that do not provide enough advanced scheduling time. For example, if your component receives a `DataHScheduleData` request that it cannot satisfy, it can fail the request with an appropriate error code.

SEE ALSO

Client programs queue read requests by calling your component's `DataHScheduleData` function.

Writing Movie Data

As with reading movie data, data handlers provide two distinct write facilities. The `DataHPutData` function is a simple synchronous interface that allows applications to append data to the end of a container.

The `DataHWrite` function is a more capable, asynchronous write function that is suitable for movie capture operations. As is the case with the `DataHScheduleData` function, your component calls the application's data-handler completion function when you are done with the write request.

There are several other helper functions that allow applications to prepare your data handler for a movie capture operation. The `DataHCreateFile` function asks your component to create a new container. The `DataHSetFileSize` and `DataHGetFileSize` functions work with a container's size, in bytes. The `DataHGetFreeSpace` function allows applications to determine when to make a container larger. The `DataHPreextend` function asks your component to make a container larger. Applications may call your component's `DataHGetPreferredBlockSize` function in order to determine how best to interact with your data handler.

Before writing data to a data reference, applications must call your component's `DataHOpenForWrite` function to open a write path to the container. The `DataHCloseForWrite` function closes that write path.

Note that some data handlers may not support write operations. For example, some shared devices, such as a CD-ROM "jukebox," may be read-only devices. As a result, it is very important that your data handler correctly report its write capabilities to client programs. See "Selecting a Data Handler" (page 440) for information about the functions that client programs use to interrogate your data handler. For more information on writing movie data, see "Storing Movie Data" (page 443).

DataHOpenForWrite

Your component opens its current data reference for write-only access.

```
pascal ComponentResult DataHOpenForWrite (DataHandler dh);
```

`dh`	Identifies the calling program's connection to your data handler component.

DISCUSSION

After setting your component's current data reference by calling the `DataHSetDataRef` function, client programs call the `DataHOpenForWrite` function in order to start writing to the data reference. Your component should open the data reference for write-only access. If the data reference is already open or cannot be opened, return an appropriate error code.

RESULT CODES

`dataAlreadyOpenForWrite`	–2044	Data reference already open for write

DataHCloseForWrite

Your component closes write-only access to its data reference.

```
pascal ComponentResult DataHCloseForWrite (DataHandler dh);
```

`dh`	Identifies the calling program's connection to your data handler component.

DISCUSSION

Note that a client program may close its connection to your component (by calling the Component Manager's `CloseComponent` function) without closing the write path. If this happens, your component should close the data reference before closing the connection.

RESULT CODES

`dataNotOpenForWrite`	–2043	Data reference not open for write
`dataAlreadyClosed`	–2045	This reference already closed

DataHPutData

Your component writes data to its current data reference. This is a synchronous write operation that appends data to the end of the current data reference.

```
pascal ComponentResult DataHPutData (DataHandler dh, Handle h, long
                              hOffset, long *offset, long size);
```

`dh`	Identifies the calling program's connection to your data handler component.
`h`	Specifies the handle that contains the data to be written to the data reference.
`hOffset`	Identifies the offset into the handle `h` to the data to be written.
`offset`	Contains a pointer to a long. Your component returns the offset in the data reference at which your component wrote the data.
`size`	Specifies the number of bytes to write.

DISCUSSION

The `DataHPutData` function provides a high-level write interface. This is a synchronous write operation that only appends data to the end of the current data reference. That is, the client program's execution is blocked until your component returns control from this function, and the client cannot control where the data is written. As a result, most movie-capture clients (for example, Apple's sequence grabber component) use the `DataHWrite` function to write data when creating movies.

RESULT CODES

`dataNotOpenForWrite`	–2043	Data reference not open for write

SEE ALSO

Client programs can force your component to write any cached data by calling your component's `DataHFlushData` function.

DataHWrite

Your component writes data to its current data reference. This can be a synchronous write operation or an asynchronous operation, and can write data to any location in the container.

```
pascal ComponentResult DataHWrite (DataHandler dh, Ptr data, long offset,
                            long size, DHCompletionUPP completion, long refCon);
```

`dh`	Identifies the calling program's connection to your data handler component.
`data`	Specifies a pointer to the data to be written. Client programs should lock the memory area holding this data, allowing your component's `DataHWrite` function to move memory.
`offset`	Specifies the offset (in bytes) to the location in the current data reference at which to write the data.
`size`	Specifies the number of bytes to write.
`completion`	Contains a pointer to a data-handler completion function. When your data handler finishes with the client program's write request, your component must call this routine. Be sure to call this routine even if the request fails. Your component should pass the reference constant that the client program provided with the `refCon` parameter. The client program must provide a completion routine for all asynchronous write requests. For synchronous requests, client programs should set this parameter to `nil`.

`refCon`	Contains a reference constant that your data handler component should provide to the data-handler completion function specified with the `completion` parameter. For synchronous operations, client programs should set this parameter to 0.

DISCUSSION

The `DataHWrite` function provides both a synchronous and an asynchronous write interface. Synchronous write operations work like the `DataHPutData` function—the data handler component returns control to the client program only after it has serviced the write request. Asynchronous write operations allow client programs to queue write requests. Your data handler queues the request and immediately returns control to the calling program. After your component actually writes the data, it calls the client program's data-handler completion function.

RESULT CODES

`dataNotOpenForWrite`	–2043	Data reference not open for write

SEE ALSO

Client programs can force your component to write any cached data by calling your component's `DataHFlushData` function.

DataHSetFileSize

Your component sets the size, in bytes, of the current data reference.

```
pascal ComponentResult DataHSetFileSize (DataHandler dh, long fileSize);
```

`dh`	Identifies the calling program's connection to your data handler component.
`fileSize`	Specifies the new size of the container corresponding to the current data reference, in bytes.

DISCUSSION

The `DataHSetFileSize` function is functionally equivalent to the File Manager's `SetEOF` function. If the client program specifies a new size that is greater than the current size, your component should extend the container to accommodate that new size. If the client program specifies a container size of 0, your component should free all of the space occupied by the container.

DataHGetFileSize

Your component returns the size, in bytes, of the current data reference.

```
pascal ComponentResult DataHGetFileSize (DataHandler dh, long *fileSize);
```

`dh`	Identifies the calling program's connection to your data handler component.
`fileSize`	Contains a pointer to a long. Your component returns the size of the container corresponding to the current data reference, in bytes.

DISCUSSION

The `DataHGetFileSize` function is functionally equivalent to the File Manager's `GetEOF` function.

DataHCreateFile

Your component creates a new container that meets the specifications of the current data reference.

```
pascal ComponentResult DataHCreateFile (DataHandler dh, OSType creator,
                          Boolean deleteExisting);
```

`dh`	Identifies the calling program's connection to your data handler component.

`creator`	Specifies the creator type of the new container. If the client program sets this parameter to 0, your component should choose a reasonable value (for example, `'TVOD'`, the creator type for Apple's movie player).
`deleteExisting`	Indicates whether to delete any existing data. If this parameter is set to `true` and a container already exists for the current data reference, your component should delete that data before creating the new container. If this parameter is set to `false`, your component should preserve any data that resides in the container defined by the current data reference (if there is any).

DataHGetPreferredBlockSize

The `DataHGetPreferredBlockSize` function allows your component to report the block size that it prefers to use when accessing the current data reference.

```
pascal ComponentResult DataHGetPreferredBlockSize (DataHandler dh,
                          long *blockSize);
```

`dh`	Identifies the calling program's connection to your data handler component.
`blockSize`	Contains a pointer to a long. Your component returns the size of blocks (in bytes) it prefers to use when accessing the current data reference.

DISCUSSION

Different devices use different file system block sizes. This function allows your component to report its preferred block size to the client program. Note that the client program is not required to use this block size when making requests. Some clients may, however, try to accommodate your component's preference.

DataHGetFreeSpace

Your component reports the number of bytes available on the device that contains the current data reference.

```
pascal ComponentResult DataHGetFreeSpace (DataHandler dh, unsigned
                        long *freeSize);
```

`dh`	Identifies the calling program's connection to your data handler component.
`freeSize`	Contains a pointer to an unsigned long. Your component returns the number of bytes of free space available on the device that contains the container referred to by the current data reference.

DataHPreextend

Your component allocates new space for the current data reference, enlarging the container.

```
pascal ComponentResult DataHPreextend (DataHandler dh, unsigned
                        long maxToAdd, unsigned long *spaceAdded);
```

`dh`	Identifies the calling program's connection to your data handler component.
`maxToAdd`	Specifies the amount of space to add to the current data reference, in bytes. If the client program sets this parameter to 0, your component should add as much space as it can.
`spaceAdded`	Contains a pointer to an unsigned long. Your component returns the number of bytes it was able to add to the data reference, in bytes.

DISCUSSION

This function is essentially analogous to the File Manager's `PBAllocContig` function. Your component should allocate contiguous free space. If there is not sufficient contiguous free space to satisfy the request, your component should return a `dskFulErr` error code.

Client programs use this function in order to avoid incurring any space-allocation delay when capturing movie data.

Managing Data Handler Components

Your data handler component provides a number of functions that applications can use to manage their connections to your handler. The most important among these is `DataHTask`, which provides processor time to your handler. Applications should call this function often so that your handler has enough time to do its work.

Applications may call your handler's `DataHPlaybackHints` function in order to provide you with some guidelines about how those applications play to use the current data reference.

The `DataHFlushData` and `DataHFlushCache` functions allow applications to influence how your component manages its stored data.

For more information on managing data handlers, see "Managing the Data Handler" (page 444).

DataHTask

Client programs call your component's `DataHTask` function in order to cede processor time to your data handler.

```
pascal ComponentResult DataHTask (DataHandler dh);
```

`dh`	Identifies the calling program's connection to your data handler component.

DISCUSSION

This function is essentially analogous to the Movie Toolbox's `MoviesTask` function. Client programs call this function in order to give your data handler component time to do its work. Your data handler uses this time to do its work. Because client programs will call this function frequently, and especially so during movie playback or capture, your data handler should return control quickly to the client program.

DataHFlushCache

Your component discards the contents of any cached read buffers.

```
pascal ComponentResult DataHFlushCache (DataHandler dh);
```

`dh`	Identifies the calling program's connection to your data handler component.

DISCUSSION

Client programs may call this function if they have, in some way, changed the container associated with the current data reference on their own. Under these circumstances, data your component may have read and cached in anticipation of future read requests from the client may be invalid.

Note that this function does not invalidate any queued read requests (made by calling your component's `DataHScheduleData` function).

DataHFlushData

Your component forces any data in its write buffers to be written to the device that contains the current data reference.

```
pascal ComponentResult DataHFlushData (DataHandler dh);
```

`dh`	Identifies the calling program's connection to your data handler component.

DISCUSSION

This function is essentially analogous to the File Manager's `PBFlushFile` function. The client program may call this function after any write operation (either `DataHPutData` or `DataHWrite`). Your component should do what is necessary to make sure that the data is written to the storage device that contains the current data reference.

DataHPlaybackHints

The `DataHPlaybackHints` function allows the client program to provide additional information to your component that you may use to optimize the operation of your data handler.

```
pascal ComponentResult DataHPlaybackHints (DataHandler dh, long flags,
                        unsigned long minFileOffset, unsigned long
                        maxFileOffset, long bytesPerSecond);
```

`dh`	Identifies the calling program's connection to your data handler component.
`flags`	Reserved for use by Apple Computer, Inc. Client programs should always set this parameter to 0.
`minFileOffset`	Together with the `maxFileOffset` parameter, specifies the range of data the client program anticipates using from the current data reference. This parameter specifies the earliest byte the program expects to use (that is, the minimum container offset value). If the client expects to access bytes from the beginning of the container, it should set this parameter to 0.
`maxFileOffset`	Specifies the latest byte the program expects to use (that is, the maximum container offset value). If the client expects to use bytes throughout the container, the client should set this parameter to –1.
`bytesPerSecond`	Indicates the rate at which your data handler must read data from the data reference in order to keep up with the client program's anticipated needs.

DISCUSSION

Your component should be prepared to have this function called more than once for a given data reference. For example, the Movie Toolbox calls this function whenever a movie's playback rate changes. This is a handy way for your data handler to track playback rate changes.

Completion Function

When client programs schedule asynchronous read or write operations (by calling your component's `DataHScheduleData` or `DataHWrite` functions), they furnish your component a data-handler completion function. Your component must call this function when it completes the read or write operation, whether the operation was a success or a failure.

Data handler Completion Function

The client program's completion function must present the following interface:

```
pascal void DHCompleteProc (Ptr request, long refcon, OSErr err);
```

`request`	Specifies a pointer to the data that was associated with the read (`DataHScheduleData`) or write (`DataHWrite`) request. The client program uses this pointer to determine which request has completed.
`refcon`	Contains a reference constant that the client program supplied to your data handler component when it made the original request.
`err`	Indicates the success or failure of the operation. If the operation succeeded, set this parameter to 0. Otherwise, specify an appropriate error code.

Graphics Importer Components

This chapter describes graphics importer components. **Graphics importer components** provide a standard method for opening and displaying still images contained within graphics documents and for working with any type of image data, regardless of the file format or compression used in the graphics document.

Use the `GetGraphicsImporterForFile` function or the `GetGraphicsImporterForDataRef` function to obtain a graphics importer component instance for a particular file. Then you can use that component instance to find out the image file's characteristics and draw it. When you are done with the component instance, call `CloseComponent`.

Most still image file formats define both how images should be stored and compressed. However, two of the file formats supported by QuickTime are container formats, which describe storage mechanisms independent of compression. These formats are QuickDraw Picture (PICT) files and QuickTime Image (QTIF) files.

QuickTime has permitted compressed image data to be included in QuickDraw pictures since QuickTime 1.0. However, the technical challenges of parsing, interpreting and spooling picture files can make them a discouraging choice for applications which are primarily interested in accessing the compressed data inside.

The QuickTime Image file format provides a much simpler container for QuickTime compressed still images. The format uses the same atom-based structure as a QuickTime movie. Because the QuickTime Image file is a single fork format, it works well in cross-platform applications. On Mac OS systems, QuickTime Image files are identified by the file type `'qtif'`. On other platforms, Apple recommends that you use the filename extension .QIF to identify QuickTime Image files.

About Graphics Importer Components

Graphics importer components, introduced in QuickTime 2.5, provide a standard method for applications to open and display still images contained within graphics documents. Graphics importer components allow you to work with any type of image data, regardless of the file format or compression used in the document. You specify the document that contains the image, and the destination rectangle the image should be drawn into, and QuickTime handles the rest. More complex interactions are also supported.

The following example code (Listing 16-1) shows the basic functions you use to draw an image file.

Listing 16-1 The basic functions used to draw an image file

```
void drawFile(const FSSpec *fss, const Rect *boundsRect)
    {
        GraphicsImportComponent gi;
        GetGraphicsImporterForFile(fss, &gi);
        GraphicsImportSetBoundsRect(gi, boundsRect);
        GraphicsImportDraw(gi);
        CloseComponent(gi);
    }
```

The same code can be used to display any image, regardless of the file format.

Supported Image File Formats

QuickTime 2.5 supports the following image file formats: QuickDraw PICT, QuickTime Image, MacPaint, Photoshop (versions 2.5 and 3.0), Silicon Graphics, GIF, JFIF/JPEG and QuickDraw GX Picture (if QuickDraw GX is installed). QuickTime 3 adds support for BMP, PNG, Targa, and TIFF. For more information, see the *QuickTime Image File Format Specificationi* , which describes the import formats.

Third-party developers may also provide graphics importers for other image file formats. Future versions of QuickTime may include support for new formats.

QuickTime Image File Format

QuickTime's ability to include any compressed image data in a QuickDraw picture is a helpful feature from a compatibility perspective. However, it presents several technical challenges for applications that need to work with compressed image data contained within pictures. Determining if compressed data is present, and extracting it, requires special code installed in QuickDraw bottlenecks to detect and copy compressed data as it processes. Additional problems are posed by special cases such as multiple compressed images in a single file. The QuickTime Image file (QTIF) format solves this and other issues.

QuickTime Image files are intended to provide the most useful container for QuickTime compressed still images. The format uses the same atom-based structure as a QuickTime movie. (See Chapter 1, "Movie Toolbox," in this reference guide for information about atoms.)

Atom Types in QuickTime Image Files

There are two defined atom types: `'idsc'`, which contains an image description, and `'idat'`, which contains the image data. This is illustrated in Figure 16-1. For a JPEG image, the image description atom contains a QuickTime image description describing the JPEG image's size, resolution, depth, and so on, and the image data atom contains the actual JPEG compressed data, as shown in Table 16-1.

Figure 16-1 An ‘idsc’ atom followed by an ‘idat’ atom

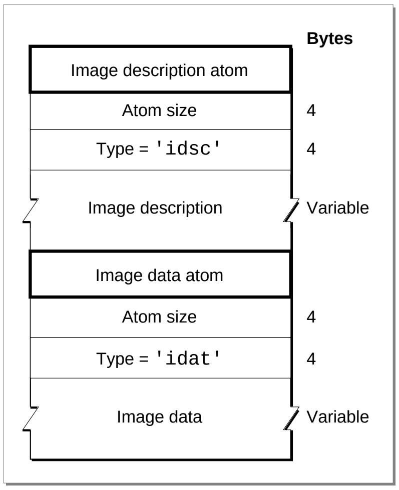


Table 16-1 A QuickTime Image file containing JPEG compressed data

`0000005E`	Atom size, 94 bytes
`69647363`	Atom type, `'idsc'`
`00000056`	Image description size, 86 bytes
`6A706567`	Compressor identifier, `'jpeg'`
`00000000`	Reserved, set to zero
`0000`	Reserved, set to zero
`0000`	Reserved, set to zero
`00000000`	Major and minor version of this data, zero if not applicable
`6170706C`	Vendor who compressed this data, `'appl'`
`00000000`	Temporal quality, zero (no temporal compression)
`00000200`	Spatial quality, codecNormalQuality
`0140`	Image width, 320
`00F0`	Image height, 240
`00480000`	Horizontal resolution, 72 dpi
`00480000`	Vertical resolution, 72 dpi
`00003C57`	Data size, 15447 bytes (use zero if unknown)
`0001`	Frame count, 1

`0C 50 68 6F 74 6F 20 2D`
`20 4A 50 45 47 00 00 00`
`00 00 00 00 00 00 00 00`
`00 00 00 00 00 00 00 00`
	Compressor name, "Photo - JPEG" (32 byte Pascal string)

`0018`	Image bit depth, 24
`FFFF`	Color look-up table ID, -1 (none)
`00003C5F`	Atom size, 15455 bytes
`69646174`	Atom type, `'idat'`

`FF D8 FF E0 00 10 4A 46`
`49 46 00 01 01 01 00 48`
`...`
	JPEG compressed data

A QuickTime Image file can also contain other atoms. For example, it can contain single-fork preview atoms.

IMPORTANT
The exact order and size of atoms is not guaranteed to match the example in Table 16-1. Applications reading QuickTime image files should always use the atom size to traverse the file and ignore atoms of unrecognized types. ▲

Note
Like QuickTime movie files, QuickTime Image files are big-endian. However, image data is stored in the same byte order as usually specified by the particular compression format. ◆

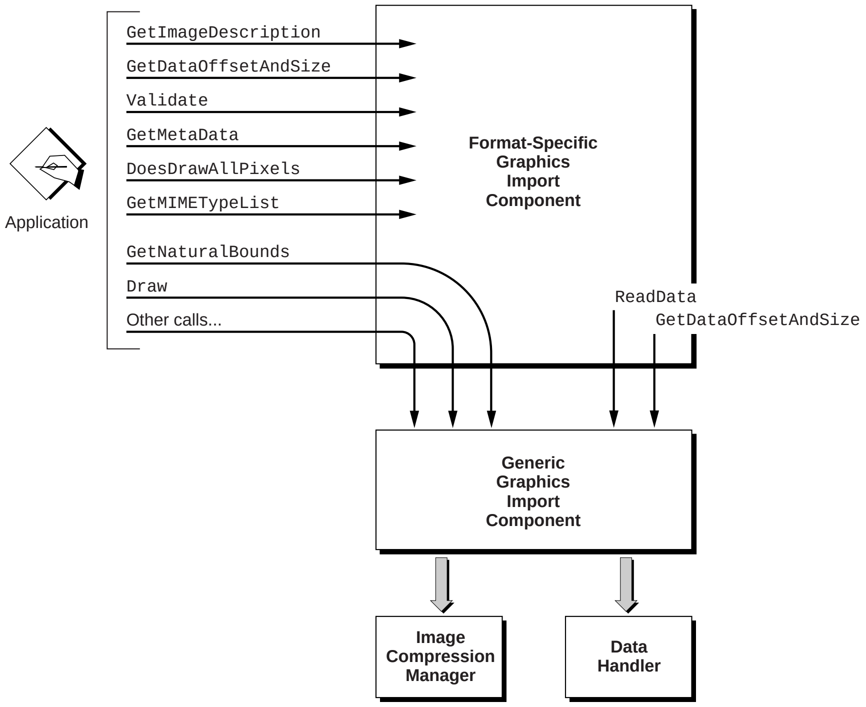
Recommended File Type and Suffix

Because the QuickTime Image file is a single-fork format, it works well in cross-platform applications. On Mac OS systems, QuickTime Image files are identified by the file type `'qtif'`. Apple recommends using the filename extension .QIF to identify QuickTime Image files on other platforms.

How Graphics Import Components Work

Format-specific graphics import components, such as the importers for JPEG, PNG, TIFF etc., are simple components. When a format-specific graphics importer is opened, it opens and targets an instance of the generic importer. Subsequently, it delegates most of its calls to the generic importer instance, as shown in Figure 16-2.

Figure 16-2 Delegating calls to the generic importer



The generic importer communicates with data handler components to negotiate access to image file data, and with the Image Compression Manager to arrange for image rendering. The only service a format-specific importer must provide is the `GraphicsImportGetImageDescription` call, which examines an image file and constructs an image description for it. The generic importer uses this image description to respond to other calls such as `GraphicsImportGetNaturalBounds` and `GraphicsImportDraw`. (In the case of `GraphicsImportDraw`, the image description is passed to the Image Compression Manager, so the `cType` field must identify a codec that will be able to draw the image. If a graphics importer needs to pass extra information that the codec will need at `PreDecompress` time, it can pass it in an image description extension.)

Graphics import components may override other calls, such as `GraphicsImportGetMetaData`, which extracts supplemental information from an image file, and `GraphicsImportGetMIMETypeList`, which provides information about the format.

Another optional call is `GraphicsImportValidate`, which attempts to ascertain quickly whether a file matches the importer's format. This is especially useful for formats which start with identifying codes or "magic numbers" (such as PNG and TIFF) in situations where image files do not have helpful file types or suffixes. In situations like this, the Image Compression Manager may ask many graphics importers in turn to validate until it finds one that accepts the file, so it is important that `GraphicsImportValidate` calls not be too slow.

Graphics importers supporting image formats which can have transparent regions should implement the `GraphicsImportDoesDrawAllPixels` call so as to warn applications that they may need to erase the destination area before drawing.

Registering Graphics Import Components

Graphics import components have component type `'grip'`. The interpretation of the subtype depends on the `movieImportSubTypeIsFileExtension` component flag. If this flag is clear, the subtype is a Macintosh file type. If this flag is set, the subtype is a file name suffix; it should be in uppercase and followed by space characters to pad it out to four characters. For instance, the file name suffix ".png" would be represented by the subtype `'PNG '`.

It is often useful to register graphics import components multiple times, so that both the file type and file name suffix may be matched. An efficient way to do this is to register the second and subsequent components as component aliases to the first. For more information about component aliases, see Chapter 2, "Component Manager."

Graphics import components that use the generic importer's `Draw` method should set the `graphicsImporterUsesImageDecompressor` flag in their component flags.

Using Graphics Importer Components

This section describes how you can obtain graphics importer component instances and use them to draw and manipulate image files.

Obtaining Graphics Import Components

You can use the `GetGraphicsImporterForFile` function to open a suitable graphics import component for a file. (If you have a data reference rather than a file, you can use the `GetGraphicsImporterForDataRef` function instead.)

When the Image Compression Manager's `GetGraphicsImporterForFile` function searches for a graphics import component, it tries, in order:

- matching the MacOS file type
- matching the file name suffix
- matching the MIME type and MIME suggested file name suffix
- asking individual graphics importer components if the file's contents match their format. It only does this for those graphics importers which support the `GraphicsImporterValidate` call.

The last stage can be time-consuming, since it involves opening many components in turn. If you want to skip it, call `GetGraphicsImporterForFileWithFlags` (or `GetGraphicsImporterForDataRefWithFlags`) and pass the `kDontUseValidateToFindGraphicsImporter` flag.

Note
If you include `kQTFileTypeQuickTimeImage` (`'qtif'`) in the list of types passed to `StandardGetFilePreview`, all files that can be opened with graphics importers are included in the file list. (The slow validate approach is not used in this case.) ◆

When you are done with the graphics importer instance, you call `CloseComponent`.

Note
If you expect to draw the same image more than once, you can improve performance by keeping the graphics importer component open, rather than creating and disposing it each time. ◆

Once you have a graphics import component for your file or data reference, you can interrogate it to determine the properties of the file, configure drawing parameters, and draw or export the file. The next two sections explain how to do this.

Determining the Properties of the Image File

If you want to know the dimensions of the image, call the importer's `GraphicsImportGetNaturalBounds` function. If you want to know other information that is represented in the image description, such as its depth or color table, call `GraphicsImportGetImageDescription`. If you want to extract meta-data from the image file (such as textual comments, like copyright information), call `GraphicsImportGetMetaData`. If you want to know information about the file format, such as the MIME types and MIME suggested file name suffixes, call `GraphicsImportGetMIMETypeList`.

Some image file formats can contain transparent regions, and hence may leave some pixels in their rectangular range unmodified even when drawing with a copying transfer mode and an identity matrix. If you would like to know whether a graphics importer's format supports such transparent regions, call `GraphicsImportDoesDrawAllPixels`. (If a graphics importer doesn't support the `GraphicsImportDoesDrawAllPixels` call, you can assume it will draw all pixels.)

Drawing and Converting Image Files

Before drawing, you may wish to set various parameters. Among those you can configure are

- the source rectangle
- transformation matrix
- clipping region
- graphics transfer mode
- drawing quality
- the destination graphics port and device

These parameters are explained in detail in Table 16-2.

Table 16-2 Drawing parameters you may configure

Parameters	Description
source rectangle	Used to display a rectangular portion of the compressed image.
transformation matrix	Used to shift, scale, rotate, and apply perspective to the source portion of the image. (Set this with `setMatrix` or `setBoundsRect`.)
clipping region	Used to restrict the area to be drawn.
graphics transfer mode	Used to define how source pixels modify destination pixels.
drawing quality	Used to specify quality vs. time tradeoffs—for example, if you're drawing a JPEG image to an 8-bit color screen, the drawing quality determines whether a slower or faster dither will be used.
destination graphics port and device	Defines the graphics environment for drawing.

Once you have set drawing parameters, you can call `Draw` to draw the image.

You can also call `GraphicsImportGetAsPicture` to get the image in the form of a QuickDraw picture handle, `GraphicsImportSaveAsPicture` to save it in a PICT file, or `GraphicsImportSaveAsQuickTimeImage` to save it in a QuickTime Image file. To export it to other image file formats, you can use `GraphicsImportExportImageFile`, or `DoExportImageFileDialogDoExportImageFileDialog` to present a standard "Export As..." dialog box.

When you are done with a graphics import component, you call `CloseComponent`.

Graphics Importer Components Reference

Data Types

Graphics Importer Component Type

```
typedef ComponentInstance GraphicsImportComponent;

enum {
    GraphicsImporterComponentType = 'grip'
};
```

MIME Type List

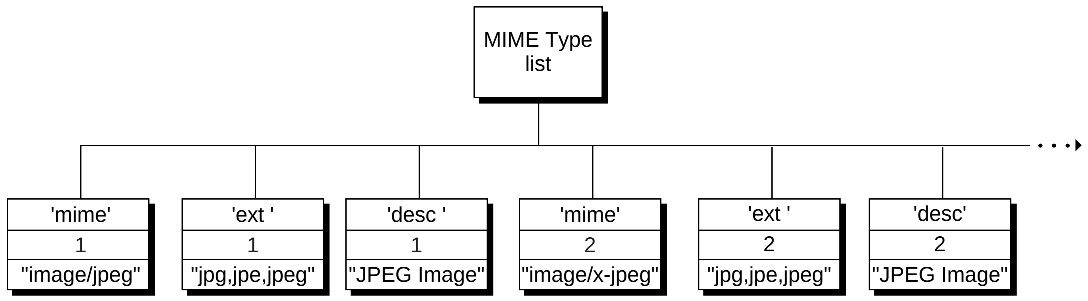
The `GraphicsImportGetMIMETypeList` function returns a list of the MIME types supported by a graphics importer component. This list is contained in the QT atom container described in this section. For more information about QT atom containers, see "QuickTime Atoms" (page 47).

At the top level of the atom container are three atoms for each supported MIME type. The atoms whose IDs are 1 describe the first supported MIME type, the atoms whose IDs are 2 describe the second supported MIME type, and so on. Note that the IDs have to be consecutive.

- An atom of type `kMimeInfoMimeTypeTag` contains a string that identifies the MIME type, such as `image/jpeg` or `image/x-jpeg`.
- The atom of type `kMimeInfoFileExtensionTag` contains a string that specifies likely file extensions for files of this MIME type, such as `jpg`, `jpe`, and `jpeg`. If there is more than one extension, the extensions are separated by commas.
- The atom of type `kMimeInfoDescriptionTag` contains a string describing the MIME type for end users, such as `JPEG Image`. These are neither Pascal nor C strings—just ASCII characters.

Figure 16-3 illustrates a MIME type list.

Figure 16-3 A MIME type list



Functions

Obtaining a Graphics Importer Instance

GetGraphicsImporterForFile

Locates and opens a graphics importer component that can be used to draw the specified file.

```
pascal OSErr GetGraphicsImporterForFile(
                    const FSSpec *theFile,
                    ComponentInstance *gi);

pascal OSErr GetGraphicsImporterForFileWithFlags(
                    const FSSpec *theFile,
                    ComponentInstance *gi,
                    long flags);
```

`theFile`	Specifies the file to be drawn using a graphics importer component.

`gi`	A pointer to a ComponentInstance in which the best graphics importer for working with the specified file will be returned. If no graphics importer can be found, the ComponentInstance will be set to `nil`.
`flags`	Controls the graphics importer search process. The following flag is available:

`kDontUseValidateToFindGraphicsImporter`
: If a graphics importer cannot be found using the file's type or file name suffix, give up immediately without using the slower process of asking every graphics importer to validate the file.

DISCUSSION

`GetGraphicsImporterForFile` first tries to locate a graphics importer component for the specified file based on the Macintosh file type of the file. If it is unable to locate a graphics importer component based on the Macintosh file type, and the file type is `'TEXT'`, `GetGraphicsImporterForFile` will try to locate a graphics importer component for the specified file based on the file name extension of the file.

If it is unable to locate a graphics importer for the file, the returned `ComponentInstance` is set to `nil`. If it is able to locate a graphics importer for the file, the returned graphics importer `ComponentInstance` will have already been set up to draw the specified file in the current port.

The caller of `GetGraphicsImporterForFile` is responsible for closing the returned `ComponentInstance` using `CloseComponent`.

GetGraphicsImporterForDataRef

Locates and opens a graphics importer component that can be used to draw the specified data reference.

```
pascal OSErr GetGraphicsImporterForDataRef(
                    Handle dataRef,
                    OSType dataRefType,
                    ComponentInstance *gi);

pascal OSErr GetGraphicsImporterForDataRefWithFlags(
                    Handle dataRef,
                    OSType dataRefType,
                    ComponentInstance *gi,
                    long flags);
```

`dataRef`	Specifies the data reference to be drawn using a graphics importer component.
`dataRefType`	The type of the data reference specified by the dataRef parameter. For alias-based data references, the `dataRef` handle contains an AliasRecord, and `dataRefType` is equal to `rAliasType`.
`gi`	A pointer to a `ComponentInstance` in which the best graphics importer for working with the specified data reference will be returned. If no graphics importer can be found, the `ComponentInstance` will be set to `nil`.
`flags`	Controls the graphics importer search process. The following flag is available:

`kDontUseValidateToFindGraphicsImporter`
: If a graphics importer cannot be found using the file's type, file name suffix or MIME type, give up immediately without using the slower process of asking every graphics importer to validate the file.

DISCUSSION

`GetGraphicsImporterForDataRef` tries to locate a graphics importer component for the specified data reference by checking the file name extension of the file, the file type information, and the MIME type of the file. The file name extension is retrieved from the data reference by use of the `DataHGetFileName` call to the data handler associated with the data reference.

If it is unable to locate a graphics importer for the file, the returned ComponentInstance is set to `nil`. If it is able to locate a graphics importer for the data reference, the returned graphics importer `ComponentInstance` will have already been set up to draw the specified data reference in the current port.

The caller of `GetGraphicsImporterForDataRef` is responsible for closing the returned `ComponentInstance` using `CloseComponent`.

Getting Image Characteristics

These functions are called by applications to obtain information about images.

Note
Format-specific graphics importers always implement `GraphicsImportGetImageDescription` and may optionally implement the remaining functions in this section. ◆

GraphicsImportGetNaturalBounds

Returns the bounding rectangle of an image.

```
extern pascal ComponentResult GraphicsImportGetNaturalBounds(
              GraphicsImportComponent ci,
              Rect *naturalBounds);
```

`ci`	Specifies the component instance that identifies your connection to the graphics importer component.
`naturalBounds`	A pointer to a rectangle structure describing the size of the bounding rectangle for the image.

DISCUSSION

You can use the `GraphicsImportGetNaturalBounds` function to determine the native size of the image associated with a graphics importer component. The natural bounds are always zero-based. This is a convenience function that simply calls `GraphicsImportGetImageDescription` and extracts the width and height fields.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memFullErr`	–108	Not enough memory available

GraphicsImportGetImageDescription

Returns image description information.

```
extern pascal ComponentResult GraphicsImportGetImageDescription (
                    GraphicsImportComponent ci,
                    ImageDescriptionHandle *desc);
```

`ci`	Specifies the component instance that identifies your connection to the graphics importer component.
`desc`	A handle to an image description structure.

DISCUSSION

The `GraphicsImportGetImageDescription` function returns an image description structure containing information such as the format of the compressed data, its bit depth, natural bounds, and resolution. The caller is responsible for disposing of the returned image description handle.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memFullErr`	–108	Not enough memory available

GraphicsImportGetDataOffsetAndSize

Returns the offset and size of the compressed image data within a file.

```
extern pascal ComponentResult GraphicsImportGetDataOffsetAndSize (
                        GraphicsImportComponent ci,
                        unsigned long *offset,
                        unsigned long *size);
```

`ci`	Specifies the component instance that identifies your connection to the graphics importer component.
`offset`	A pointer to a value describing the byte offset of the image data from the beginning of the data source.
`size`	A pointer to a value describing the size of the image data in bytes.

DISCUSSION

This function returns the offset and size of the actual image data within the data source. By default, the offset returned is 0 and the size returned is the size of the file. However, some graphics import components will override this function to skip over unneeded information at the beginning or end of the file.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memFullErr`	–108	Not enough memory available

GraphicsImportValidate

Validates image data for a data reference.

```
pascal ComponentResult GraphicsImportValidate (
                        GraphicsImportComponent ci,
                        Boolean *valid);
```

`ci`	Specifies the component instance that identifies your connection to the graphics importer component.
`valid`	Pointer to a Boolean value. On return, this parameter is set to `true` if the the graphics importer component can draw the data reference. If the graphics importer component cannot draw the data reference, this parameter is set to `false`.

DISCUSSION

The `GraphicsImportValidate` functions allows a graphics importer component to determine if its current data reference contains valid image data. For example, a JFIF graphics importer component might check for the presence of a JFIF marker at the start of the data stream. This function is provided for applications to use to determine what type of image data a particular file may contain. Sometimes a file may not have the correct file type or file extension. In this case, the application will not know which graphics importer component to use. By iterating through all graphics importer components and calling `GraphicsImportValidate` for each one, it may be possible to locate a graphics importer component that can draw the specified file.

Not all graphics importer components implement this function. A component that does not implement the function will return the `badComponentSelector` result code. This does not indicate that the file is valid or invalid.

Note
`GraphicsImportValidate` does not perform an exhaustive sanity check on the file. It is possible for `GraphicsImportValidate` to claim a data reference is valid but for `GraphicsImportDraw` to return an error due to bad data. ◆

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified.
`memFullErr`	–108	Not enough memory available
`badComponentSelector`	0x80008002	Component does not support the specified request code

GraphicsImportGetMetaData

Extracts metadata from an image file.

```
extern pascal ComponentResult GraphicsImportGetMetaData (
                        GraphicsImportComponent ci,
                        void *userData);
```

`ci`	Specifies the component instance that identifies your connection to the graphics importer component.
`userData`	Contains a pointer to a user data structure. The value you pass should have parameter type `UserData`.

DISCUSSION

This function extracts metadata from an image file and adds it to a user data structure. (Note that `userData` must already be allocated.) Different image file formats support different kinds of metadata and have different ways of identifying them. For more information about the kinds of metadata extracted by QuickTime's graphics import components, see the *QuickTime Image File Format Specification*.

You may create a new user data structure by calling `NewUserData`. Alternatively, you can obtain a pointer to an existing one by calling `GetMovieUserData`, `GetTrackUserData` or `GetMediaUserData`. If the user data passed to `GraphicsImportGetMetaData` belongs to a movie, track or media, then whatever metadata is extracted will be added to that movie, track or media.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified.
`memFullErr`	–108	Not enough memory available
`badComponentSelector`	0x80008002	Component does not support the specified request code

GraphicsImportDoesDrawAllPixels

Asks whether the graphics importer expects to draw every pixel.

```
extern pascal ComponentResult GraphicsImportDoesDrawAllPixels (
                    GraphicsImportComponent ci,
                    short *drawsAllPixels);
```

`ci`	Specifies the component instance that identifies your connection to the graphics importer component.
`drawsAllPixels`	A pointer to a value describing the predicted drawing behavior.

DISCUSSION

Some image file formats permit non-rectangular images or images with transparent regions. When such an image is drawn, not every pixel in the boundary rectangle will be changed. The `GraphicsImportDoesDrawAllPixels` function lets you try to find out whether this will be the case. For instance, you might choose to erase the area behind the image before drawing.

If the graphics import component supports this function, `drawsAllPixels` will contain one of the following values on return:

`graphicsImporterDrawsAllPixels`	0	Every pixel in the boundary rectangle will be drawn.
`graphicsImporterDoesntDrawAllPixels`	1	Some pixels in the boundary rectangle will not be drawn.
`graphicsImporterDontKnowIfDrawAllPixels`	2	The graphics importer cannot determine whether all pixels will be drawn.

If the graphics import component does not support this function, it will return `badComponentSelector`. In this case, it is usually safe to assume that the graphics importer will draw all pixels.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified.
`memFullErr`	–108	Not enough memory available
`badComponentSelector`	0x80008002	Component does not support the specified request code

Setting Drawing Parameters

The functions described in this section allow you to specify various parameters for drawing operations, such as clipping, scaling, graphics mode, anddecompression quality. All of these functions are based on corresponding routines in the Image Compression Manager for working with image decompression sequences.

GraphicsImportSetBoundsRect

Defines the rectangle in which to draw an image.

```
extern pascal ComponentResult GraphicsImportSetBoundsRect (
              GraphicsImportComponent ci,
              const Rect *bounds);
```

`ci`	Specifies the component instance that identifies your connection to the graphics importer component.
`bounds`	A pointer to a rectangle structure describing the bounding rectangle into which the image will be drawn.

DISCUSSION

You use this function to define the rectangle into which the graphics image should be drawn. The function creates a transformation matrix to map the image’s natural bounds to the specified bounds and then calls the `GraphicsImportSetMatrix` function.

Note
Because this function affects the transformation matrix, you should use the `GraphicsImportSetMatrix` function (page 504) instead of this function when you also need to specify more complex transformations of the matrix. ◆

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memFullErr`	–108	Not enough memory available

GraphicsImportGetBoundsRect

Returns the bounding rectangle for drawing.

```
extern pascal ComponentResult GraphicsImportGetBoundsRect (
                     GraphicsImportComponent ci,
                     Rect *bounds);
```

`ci`	Specifies the component instance that identifies your connection to the graphics importer component.
`bounds`	A pointer to a rectangle structure describing the bounding rectangle that has been defined for the image.

DISCUSSION

This is a convenience function that is implemented by calling `GraphicsImportGetMatrix` (page 505) and `GraphicsImportGetNaturalBounds` (page 496) and using the results to calculate the drawing rectangle.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memFullErr`	–108	Not enough memory available

GraphicsImportSetMatrix

Defines the transformation matrix to use for drawing an image.

```
extern pascal ComponentResult GraphicsImportSetMatrix (
                        GraphicsImportComponent ci,
                        const MatrixRecord *matrix);
```

`ci`	Specifies the component instance that identifies your connection to the graphics importer component.
`matrix`	A pointer to a matrix structure that specifies how to transform the image during decompression. For example, you can use a transformation matrix to scale or rotate the image. To set the matrix to identity, pass `nil` in this parameter.

DISCUSSION

The `GraphicsImportSetMatrix` function establishes the transformation matrix to be applied to an image, which determines where and how it will be drawn.

Note

This function affects the bounding rectangle defined for the image. You can specify where an image will be drawn by setting either a transformation matrix or a bounding rectangle, but it is usually more convenient for applications to set a bounding rectangle using the `GraphicsImportSetBoundsRect` (page 502) function. ◆

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memFullErr`	–108	Not enough memory available

SEE ALSO

For more information about transformation matrices and the functions for working with them, see the Movie Toolbox chapter of *Inside Macintosh: QuickTime*.

GraphicsImportGetMatrix

Returns the transformation matrix to be used for drawing.

```
extern pascal ComponentResult GraphicsImportGetMatrix (
                    GraphicsImportComponent ci,
                    MatrixRecord *matrix);
```

`ci`	Specifies the component instance that identifies your connection to the graphics importer component.
`matrix`	A pointer to the transformation matrix that has been defined for the image.

DISCUSSION

The transformation matrix is initialized to the identity matrix when the graphics import component is instantiated.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memFullErr`	–108	Not enough memory available

GraphicsImportSetClip

Defines the clipping region for drawing.

```
extern pascal ComponentResult GraphicsImportSetClip (
                    GraphicsImportComponent ci,
                    RgnHandle clipRgn);
```

`ci`	Specifies the component instance that identifies your connection to the graphics importer component.
`clipRgn`	A handle to a clipping region in the destination coordinate system. Set to `nil` to disable clipping. The graphics import component makes a copy of this region.

DISCUSSION

Because all drawing operations ignore the port clip, you must use this function to clip an image. The graphics importer component draws only that portion of the image that lies within the specified clipping region.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memFullErr`	–108	Not enough memory available

GraphicsImportGetClip

Returns the current clipping region.

```
extern pascal ComponentResult GraphicsImportGetClip (
                    GraphicsImportComponent ci,
                    RgnHandle *clipRgn);
```

`ci`	Specifies the component instance that identifies your connection to the graphics importer component.
`clipRgn`	A handle to the clipping region that has been defined for the image. Returns `nil` if there is no clipping region.

DISCUSSION

The caller must dispose of the returned region handle.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memFullErr`	–108	Not enough memory available

GraphicsImportSetGraphicsMode

Sets the graphics transfer mode for an image.

```
extern pascal ComponentResult GraphicsImportSetGraphicsMode (
                    GraphicsImportComponent ci,
                    long graphicsMode,
                    const RGBColor *opColor);
```

`ci`	Specifies the component instance that identifies your connection to the graphics importer component.
`graphicsMode`	Specifies the graphics transfer mode to use for drawing the image. QuickTime supports the same graphics modes as Color QuickDraw's `CopyBits` function (described in *Inside Macintosh: Imaging with QuickDraw*) as well as any mode defined by the Image Compression manager, such as alpha modes.
`opColor`	A pointer to an RGB color structure describing the color to use for blending and transparent operations.

DISCUSSION

You can use this function to specify the graphics transfer mode and color to use for blending and transparent operations.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memFullErr`	–108	Not enough memory available

GraphicsImportGetGraphicsMode

Returns the graphics transfer mode for an image.

```
extern pascal ComponentResult GraphicsImportGetGraphicsMode (
              GraphicsImportComponent ci,
              long *graphicsMode,
              RGBColor *opColor);
```

`ci`	Specifies the component instance that identifies your connection to the graphics importer component.
`graphicsMode`	A pointer to a `long` integer. The function returns the QuickDraw graphics transfer mode setting for the image. Set to `nil` if you are not interested in this information.
`opColor`	A pointer to an RGB color structure. The function returns the color currently specified for blend and transparent operations. Set to `nil` if you are not interested in this information.

DISCUSSION

You can use this function to find out the current graphics transfer mode and color to use for blending and transparent operations. The default graphics mode is `ditherCopy` and the default `opColor` is 50% gray.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memFullErr`	–108	Not enough memory available

GraphicsImportSetQuality

Sets the image quality value.

```
extern pascal ComponentResult GraphicsImportSetQuality (
              GraphicsImportComponent ci,
              CodecQ quality);
```

`ci`	Specifies the component instance that identifies your connection to the graphics importer component.

`quality`	Specifies the desired image quality for decompression. Values for this parameter are on the same scale as compression quality. See page 3-57 of *Inside Macintosh: QuickTime* for a description of the defined quality constants.

DISCUSSION

The `quality` parameter controls how precisely the decompressor decompresses the image data. Some decompressors may choose to ignore some image data to improve decompression speed.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memFullErr`	–108	Not enough memory available

GraphicsImportGetQuality

Returns the image quality value.

```
extern pascal ComponentResult GraphicsImportGetQuality (
                        GraphicsImportComponent ci,
                        CodecQ *quality);
```

`ci`	Specifies the component instance that identifies your connection to the graphics importer component.
`quality`	A pointer to the currently specified quality value.

DISCUSSION

The `quality` value indicates how precisely the decompressor will decompress the image data. Some decompressors may choose to ignore some image data to improve decompression speed. With QuickTime 3 the default quality is `codecHighQuality`.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memFullErr`	–108	Not enough memory available

GraphicsImportSetSourceRect

Sets the source rectangle to use for an image.

```
extern pascal ComponentResult GraphicsImportSetSourceRect (
                    GraphicsImportComponent ci,
                    const Rect *sourceRect);
```

`ci`	Specifies the component instance that identifies your connection to the graphics importer component.
`sourceRect`	A pointer to a rectangle defining the portion of the image to decompress. This rectangle must lie within the boundary rectangle of the source image. Set to `nil` to use the entire image.

DISCUSSION

This function provides a way to use only a portion of the source image. Set the `sourceRect` parameter to `nil` to specify that the entire image source rectangle should be used.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memFullErr`	–108	Not enough memory available

GraphicsImportGetSourceRect

Returns the current source rectangle for an image.

```
extern pascal ComponentResult GraphicsImportGetSourceRect (
                      GraphicsImportComponent ci,
                      Rect *sourceRect);
```

`ci`	Specifies the component instance that identifies your connection to the graphics importer component.
`sourceRect`	A pointer to a rectangle structure. The function returns the source rectangle currently specified for the image.

DISCUSSION

This function returns the current source rectangle, as specified by the `GraphicsImportSetSourceRect` function. The default source rectangle is the image's natural bounds.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memFullErr`	–108	Not enough memory available

GraphicsImportSetProgressProc

Installs a progress procedure to call while drawing an image.

```
extern pascal ComponentResult GraphicsImportSetProgressProc (
                      GraphicsImportComponent ci,
                      ICMProgressProcRecordPtr progressProc);
```

`ci`	Specifies the component instance that identifies your connection to the graphics importer component.
`progressProc`	Points to a progress function structure. If you pass a value of -1, you obtain a standard progress function. If you want to remove the existing progress function, pass `nil`.

DISCUSSION

This function sets a progress function that will be installed in the image decompression sequence used to draw the image.

See Chapter 1, "Movie Toolbox." for more information about progress functions.

▲ **WARNING**
If your progress function does any drawing, you should take care to set a safe graphics state before doing so, and to restore the graphics state afterwards. In particular, the current graphics device may be an offscreen device. ▲

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memFullErr`	–108	Not enough memory available

GraphicsImportGetProgressProc

Returns the current progress procedure.

```
extern pascal ComponentResult GraphicsImportGetProgressProc (
                    GraphicsImportComponent ci,
                    ICMProgressProcRecordPtr progressProc);
```

`ci`	Specifies the component instance that identifies your connection to the graphics importer component.
`progressProc`	A pointer to a progress procedure structure.

DISCUSSION

By default, graphics import components have no progress procedure.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memFullErr`	–108	Not enough memory available

Drawing Images

GraphicsImportSetGWorld

Sets the graphics port and device for drawing.

```
extern pascal ComponentResult GraphicsImportSetGWorld (
                    GraphicsImportComponent ci,
                    CGrafPtr port,
                    GDHandle gd);
```

`ci`	Specifies the component instance that identifies your connection to the graphics importer component.
`port`	Specifies the destination graphics port or graphics world. Set to `nil` to use the current port.
`gd`	Specifies the destination graphics device. Set 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 device.

DISCUSSION

The graphics world is initialized to the current port and device when the graphics importer component is opened. You can use this function to select another port or device.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memFullErr`	–108	Not enough memory available

GraphicsImportGetGWorld

Returns the current graphics port and device for drawing.

```
extern pascal ComponentResult GraphicsImportGetGWorld (
                    GraphicsImportComponent ci,
                    CGrafPtr *port,
                    GDHandle *gd);
```

`ci`	Specifies the component instance that identifies your connection to the graphics importer component.
`port`	Returns the current destination graphics port. Set to `nil` if you are not interested in this information.
`gd`	Returns the destination graphics device. Set to `nil` if you are not interested in this information.

DISCUSSION

This function returns the graphics port and device that will be used to draw the image. The graphics world is initialized to the current port and device when the graphics importer component is opened.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memFullErr`	–108	Not enough memory available

GraphicsImportDraw

Draws an image.

```
extern pascal ComponentResult GraphicsImportDraw (GraphicsImportComponent
                    ci);
```

`ci`	Specifies the component instance that identifies your connection to the graphics importer component.

DISCUSSION

This function draws the image currently in use by the graphics import component to the graphics port and device specified by the `GraphicsImportSetGWorld` function. The `GraphicsImportDraw` function takes into account all settings you have specified for the image, such as the source rectangle, clipping region, graphics mode, and image quality.

Note
The generic graphics importer's `Draw` function uses the results of `GetImageDescription` and `GetDataOffsetAndSize` to create a decompression sequence and uses it to draw the image. Other graphics importers may override this behavior. ◆

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memFullErr`	–108	Not enough memory available

Saving Image Files

Graphics import components can save data in several formats, including QuickDraw pictures and QuickTime Image files. This capability is only needed by applications that perform file format translation. Applications that only wish to draw the image can use the `GraphicsImportDraw` function.

GraphicsImportSaveAsPicture

Creates a QuickDraw picture file.

```
extern pascal ComponentResult GraphicsImportSaveAsPicture (
                        GraphicsImportComponent ci,
                        const FSSpec *fss,
                        ScriptCode scriptTag);
```

`ci` Specifies the component instance that identifies your connection to the graphics importer component.

`fss`	A pointer to the file that is to receive the image.
`scriptTag`	Specifies the script system in which the file name is to be displayed. If you have established the name and location of the file using one of the Standard File Package functions, use the script code returned in the reply record (`reply.sfScript`). Otherwise, specify the system script by setting the `scriptTag` parameter to the value `smSystemScript`. See *Inside Macintosh: Files* for more information about script specification.

DISCUSSION

This function creates a new QuickDraw picture file containing the image currently in use by the graphics import component. If possible, the image will remain in the compressed format. For example, if the image is from a JFIF file, the picture will contain compressed JPEG data.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memFullErr`	–108	Not enough memory available

File system errors

GraphicsImportSaveAsQuickTimeImageFile

Creates a QuickTime Image file.

```
extern pascal ComponentResult GraphicsImportSaveAsQuickTimeImageFile (
                GraphicsImportComponent ci,
                const FSSpec *fss,
                ScriptCode scriptTag);
```

`ci`	Specifies the component instance that identifies your connection to the graphics importer component.
`fss`	A pointer to the file that is to receive the image.

`scriptTag`	Specifies the script system in which the file name is to be displayed. If you have established the name and location of the file using one of the Standard File Package functions, use the script code returned in the reply record (`reply.sfScript`). Otherwise, specify the system script by setting the `scriptTag` parameter to the value `smSystemScript`. See *Inside Macintosh: Files* for more information about script specification.

DISCUSSION

This function creates a new QuickTime Image file containing the image currently in use by the graphics import component. If possible, the image will remain in the compressed format. For example, if the image is from a JFIF file, the QuickTime Image file will contain compressed JPEG data.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memFullErr`	–108	Not enough memory available

File system errors

GraphicsImportGetAsPicture

Creates a QuickDraw picture handle.

```
extern pascal ComponentResult GraphicsImportGetAsPicture (
                        GraphicsImportComponent ci,
                        PicHandle *picture);
```

`ci`	Specifies the component instance that identifies your connection to the graphics importer component.
`picture`	Points to a picture handle that is to receive the image.

DISCUSSION

This function creates a new QuickDraw picture handle containing the image currently in use by the graphics import component. If possible, the image will remain in the compressed format. For example, if the image is from a JFIF file, the picture will contain compressed JPEG data. It is the responsibility of the caller to dispose of the picture handle using `KillPicture`.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memFullErr`	–108	Not enough memory available

GraphicsImportExportImageFile

Saves an image in a foreign file format.

```
extern pascal ComponentResult GraphicsImportExportImageFile
                        (GraphicsImportComponent ci,
                         OSType  fileType,
                         OSType fileCreator,
                         const FSSpec *fss,
                         ScriptCodescriptTag);
```

`ci`	Specifies the component instance that identifies your connection to the graphics importer component.
`fileType`	The file type for the new image file, such as ‘`JPEG`’.
`fileCreator`	The file creator for the new image file. This parameter may be 0, in which case a default file creator for this file type is used.
`fss`	A pointer to the file that is to receive the exported image file.
`scriptTag`	Specifies the script system in which the file name is to be displayed. If you have established the name and location of the file using one of the Standard File Package functions, use the script code returned in the reply record (`reply.sfScript`).

Otherwise, specify the system script by setting the `scriptTag` parameter to the value `smSystemScript`. See *Inside Macintosh: Files* for more information about script specification.

DISCUSSION

This function creates a new file containing the image currently in use by the graphics import component. The new file is compressed in a format corresponding to the provided file type.

If a non-identity matrix has been applied to the graphics import component, this matrix is applied to the image before export. Since most image formats don't support nonzero top-left coordinates, the matrix is temporarily adjusted to ensure that the exported image's bounds have top-left coordinates at (0,0). If the matrix does not map the graphics import component's source rectangle to a rectangle, there will be extra white space left around the image.

In QuickTime 3, the supported export file types are `kQTFileTypePicture`, `kQTFileTypeQuickTimeImage`, `kQTFileTypeBMP`, `kQTFileTypeJPEG`, and `kQTFileTypePhotoShop`.

RESULT CODES

`noErr`	0	No error
`dupFNErr`	–48	File already exists
`paramErr`	–50	Invalid parameter or unrecognized export file type

File Manager and Memory Manager errors

GraphicsImportGetExportImageTypeList

Returns information about available export formats.

```
pascal ComponentResult GraphicsImportGetExportImageTypeList(
                    GraphicsImportComponent  ci,
                     void *  qtAtomContainerPtr);
```

`ci` Specifies the component instance that identifies your connection to the graphics importer component.

`qtAtomContainerPtr`
A pointer to a QT atom container to receive the type list.

DISCUSSION

This function creates and returns a QuickTime atom container containing information about the file types that can be exported by the graphics import component. It is the responsibility of the caller to dispose of this atom container.

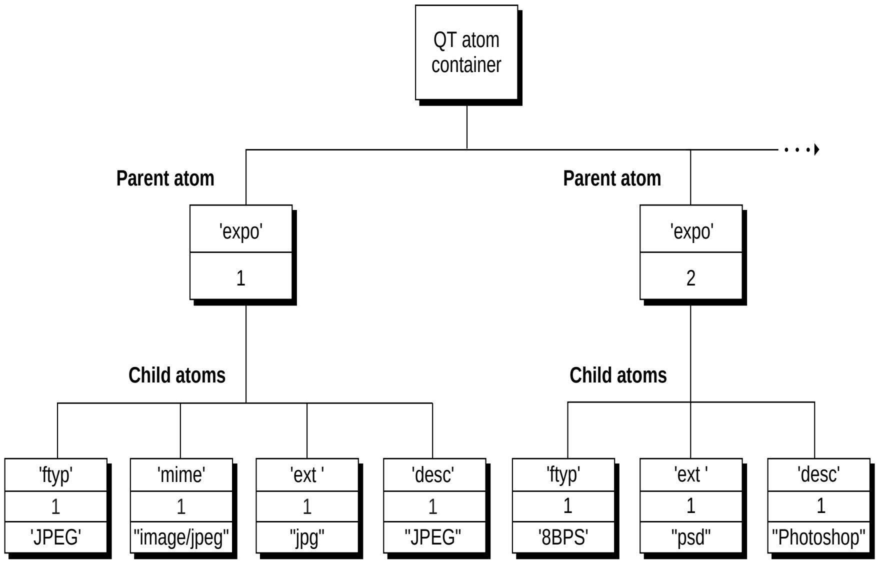
In QuickTime 3, the supported export file types are `kQTFileTypePicture`, `kQTFileTypeQuickTimeImage`, `kQTFileTypeBMP`, `kQTFileTypeJPEG`, and `kQTFileTypePhotoShop`. For each file type, the atom container contains the following child atoms:

`kGraphicsExportFileType`	The exported file type
`kGraphicsExportMIMEType`	The MIME type for this format (optional)
`kGraphicsExportExtension`	The suggested file extension for this format
`kGraphicsExportDescription`	A human-readable name for this format

Note
The file type atom should contain an OS Type; the other atoms should contain non-terminated strings. ◆

Figure 16-4 shows a diagram of a QT atom container with the following atoms grouped.

Figure 16-4 A QT atom container



RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter
`Memory Manager errors`		

GraphicsImportDoExportImageFileDialog

Presents a dialog box letting the user save an image in a foreign file format.

```
pascal ComponentResult
                    GraphicsImportDoExportImageFileDialog(
                    GraphicsImportComponent  ci,
                     const FSSpec *inDefaultSpec,
                    StringPtr prompt,
                     ModalFilterYDUPP filterProc,
                     OSType *outExportedType,
                     FSSpec *outExportedSpec,
                     ScriptCode *outScriptTag);
```

`ci`	Specifies the component instance that identifies your connection to the graphics importer component.
`inDefaultSpec`	A pointer to an `FSSpec` suggesting the default name for the file. If you do not want to suggest a default name, pass `nil`.
`prompt`	A prompt that appears in the standard put dialog box; it may be nil, in which case a default string is used.
`filterProc`	A modal filter procedure to be passed to `CustomPutFile`. If you do not need to filter events, pass `nil`.
`outExportedType`	A pointer to a variable that will receive the file type that was chosen for export. If you do not want this information, pass `nil`.
`outExportedSpec`	A pointer to a variable that will receive the file that was written. If you do not want this information, pass `nil`.
`outScriptTag`	A pointer to a variable that will receive the script system in which the exported file name is to be displayed. If you do not want this information, pass `nil`.

DISCUSSION

This function presents the user with an extended Standard File dialog box which allows the image currently in use by the graphics import component to be exported to a file and in a format of the user’s choice.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter or unrecognized export file type
`userCanceledErr`	–128	The user canceled the dialog box

File Manager and Memory Manager errors

GraphicsImportGetExportSettingsAsAtomContainer

Retrieves settings for exported image files.

```
extern pascal ComponentResult
              GraphicsImportGetExportSettingsAsAtomContainer
              (GraphicsImportComponent  ci,
               void *  qtAtomContainerPtr);
```

`ci` Specifies the component instance that identifies your connection to the graphics importer component.

`qtAtomContainerPtr`

A pointer to a QT atom container to receive the settings information.

DISCUSSION

This function creates and returns a new QuickTime atom container which holds information about how images will be saved by the `GraphicsImportExportImageFile` function. It is the responsibility of the caller to dispose of this atom container.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter
`memFullErr`	–108	Not enough memory available

GraphicsImportSetExportSettingsFromAtomContainer

Sets settings for exported image files.

```
extern pascal ComponentResult
              GraphicsImportSetExportSettingsFromAtomContainer
              (GraphicsImportComponent  ci,
               void *qtAtomContainer);
```

`ci`	Specifies the component instance that identifies your connection to the graphics importer component.
`qtAtomContainer`	A QuickTime atom container holding new settings information.

DISCUSSION

This function extracts export settings from a QuickTime atom container. These settings configure how images will be saved by the `GraphicsImportExportImageFile` function.

RESULT CODES

`noErr`	0	No error
`memFullErr`	–108	Not enough memory available.

Getting MIME Types

Your graphics import component can support MIME types that correspond to graphics formats it supports. To make a list of these MIME types available to applications or other software, it must implement the `GraphicsImportGetMIMETypeList` function described in this section.

GraphicsImportGetMIMETypeList

Returns a list of MIME types supported by the graphics import component.

```
pascal ComponentResult GraphicsImportGetMIMETypeList (
                     GraphicsImportComponent ci,
                     void *qtAtomContainerPtr);
```

`ci`	Specifies an instance of a graphics importer component.

`qtAtomContainerPtr`
A pointer to a MIME type list, a QT atom container that contains a list of MIME types supported by the graphics import component. The MIME type list structure is described in “MIME Type List” (page 537).

DISCUSSION

To indicate that your graphics import component supports this function, set the `hasMovieImportMIMEList` flag in the `componentFlags` field of the component description record.

Note
It is not necessary to set a data source before making this call, since it does not provide any file-specific information. ◆

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memFullErr`	–108	Not enough memory available
`badComponentSelector`	0x80008002	Component does not support the specified request code

Specifying the Data Source

Graphics importer components use QuickTime data handler components to obtain their data. Applications, however, will use the graphics importer component functions described in this section, rather than directly calling a data handler. These functions allow the data source to be a file, a handle, or a QuickTime data reference.

You do not need to call the functions in this section if you use one of the `GetGraphicsImporter` functions. The `GetGraphicsImporter` functions automatically set the graphics importer component’s data source. You only need to use these functions if you open the graphics importer component directly.

GraphicsImportSetDataFile

Specifies the file that the graphics data resides in.

```
extern pascal ComponentResult GraphicsImportSetDataFile (
                        GraphicsImportComponent ci,
                        const FSSpec *theFile);
```

`ci`	Specifies the component instance that identifies your connection to the graphics importer component.
`theFile`	A pointer to the file specification containing the graphics data.

DISCUSSION

The file will be opened for read access.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memFullErr`	–108	Not enough memory available

GraphicsImportGetDataFile

Returns the file that the graphics data resides in.

```
extern pascal ComponentResult GraphicsImportGetDataFile (
                        GraphicsImportComponent ci,
                        FSSpec *theFile);
```

`ci`	Specifies the component instance that identifies your connection to the graphics importer component.
`theFile`	A pointer in which to return the file containing the graphics data.

DISCUSSION

You use this function to get the file system specification record for the file that the graphics data resides in. If the data source is not a file, the function returns `paramErr`.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memFullErr`	–108	Not enough memory available

GraphicsImportSetDataHandle

Specifies the handle that the graphics data resides in.

```
extern pascal ComponentResult GraphicsImportSetDataHandle (
                        GraphicsImportComponent ci,
                        Handle h);
```

`ci`	Specifies the component instance that identifies your connection to the graphics importer component.
`h`	Specifies a handle containing graphics data.

DISCUSSION

The graphics importer component doesn't make a copy of this data. Therefore you must not dispose this handle until the graphics importer has been closed.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memFullErr`	–108	Not enough memory available.

GraphicsImportGetDataHandle

Returns the handle that the graphics data resides in.

```
extern pascal ComponentResult GraphicsImportGetDataHandle (
                    GraphicsImportComponent ci,
                    Handle *h);
```

`ci`	Specifies the component instance that identifies your connection to the graphics importer component.
`h`	A pointer to a handle to return a handle containing the graphics data.

DISCUSSION

You use this function to get the handle that the graphics data resides in. If the data source is not a handle, the function returns paramErr.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memFullErr`	–108	Not enough memory available

GraphicsImportSetDataReference

Specifies the data reference that the graphics data resides in.

```
extern pascal ComponentResult GraphicsImportSetDataReference (
                    GraphicsImportComponent ci,
                    Handle dataRef,
                    OSType dataReType);
```

`ci`	Specifies the component instance that identifies your connection to the graphics importer component.
`dataRef`	A pointer to a handle to return a QuickTime data reference.
`dataReType`	A pointer to the data reference type.

DISCUSSION

Applications typically do not use this function. The `GraphicsImportSetDataFile` and `GraphicsImportSetDataHandle` functions both call this function, with the appropriate data reference and data reference type.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memFullErr`	–108	Not enough memory available

GraphicsImportGetDataReference

Returns the data reference that the graphics data resides in.

```
extern pascal ComponentResult GraphicsImportGetDataReference (
                    GraphicsImportComponent ci,
                    Handle *dataRef,
                    OSType *dataReType);
```

`ci`	Specifies the component instance that identifies your connection to the graphics importer component.
`dataRef`	A pointer to the handle to return a QuickTime data reference.
`dataReType`	A pointer to the data reference type.

DISCUSSION

You use this function to get the data reference that the graphics data resides in. Both the `dataRef` and `dataReType` parameters may be set to `nil` if the corresponding information is not desired. The `GraphicsImportGetDataHandle` and `GraphicsImportGetDataFile` functions call `GraphicsImporGetDataReference` and then manipulate the result accordingly. The caller should disposed of the returned `dataRef`.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memFullErr`	–108	Not enough memory available

GraphicsImportSetDataReferenceOffsetAndLimit

Specifies the data reference starting offset and data size limit.

```
extern pascal ComponentResult
              GraphicsImportSetDataReferenceOffsetAndLimit (
              GraphicsImportComponent ci,
              unsigned long offset,
              unsigned long limit);
```

`ci`	Specifies the component instance that identifies your connection to the graphics importer component.
`offset`	The byte offset of the image data from the beginning of the data reference.
`limit`	The data limit. This value is the maximum offset into the data reference that data may be read from.

DISCUSSION

A data reference typically refers to an entire file. However, there are times when the data being referenced is a part of a larger file. In these cases, it is necessary to indicate where the data begins in the data reference and where it ends. This function lets you specify the starting offset and ending offset. All requests to read data are then relative to the specified offset, and are pinned to the data size, so the graphics import component cannot accidentally read off the end (or beginning) of the segment.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memFullErr`	–108	Not enough memory available

GraphicsImportGetDataReferenceOffsetAndLimit

Returns the data reference starting offset and data size limit.

```
extern pascal ComponentResult
              GraphicsImportGetDataReferenceOffsetAndLimit (
              GraphicsImportComponent ci,
              unsigned long *offset,
              unsigned long *limit);
```

`ci`	Specifies the component instance that identifies your connection to the graphics importer component.
`offset`	A pointer to a value describing the byte offset of the image data from the beginning of the data reference.
`limit`	A pointer to a value describing the data size limit.

DISCUSSION

This function returns the values set by the `GraphicsImportSetDataReferenceOffsetAndLimit` function. By default, offset is 0 and limit is `MaxInt` (2^32 – 1).

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memFullErr`	–108	Not enough memory available

Retrieving Image Data

This function is used by format-specific graphics import components to read data from the data source. It is implemented by the generic graphics importer.

GraphicsImportReadData

Reads image data.

```
extern pascal ComponentResult GraphicsImportReadData (
                    GraphicsImportComponent ci,
                    void *dataPtr,
                    unsigned long dataOffset,
                    unsigned long dataSize);
```

`ci`	Specifies the component instance that identifies your connection to the graphics importer component.
`dataPtr`	A pointer to a memory block to receive the data.
`dataOffset`	The offset of the image data within the data reference. The function begins reading image data from this offset.
`dataSize`	The number of bytes of image data to read.

DISCUSSION

`GraphicsImportReadData` communicates with the appropriate data handler to retrieve image data. Typically, only developers of graphics importer components will need to use this function. This function should always be used to retrieve data from the data source, rather than reading it directly.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memFullErr`	–108	Not enough memory available

QuickTime Settings Control Panel

This chapter discusses new features that have been added to the QuickTime Settings control panel.

New Features of the Control Panel

CD-ROM AutoStart

The CD-ROM AutoStart feature introduced with QuickTime 2.0 allows you to create CD-ROMs that automatically launch an application when the disc is inserted. This is useful for entertainment and educational titles because it makes it easier for users to begin using the software. QuickTime provides the low-level support necessary to recognize and launch AutoStart-enabled discs.

To create an AutoStart-enabled disc, you specify a document or application file as the AutoStart file. If the file you specified is an application, you may set it to invisible. Documents may not be invisible. If the AutoStart file is a document, QuickTime asks the Finder to launch the document. If an application is not available, the Finder will issue its normal warning, just as if the file had been double-clicked.

AutoStart works only with HFS discs. All information about AutoStart is contained in sector 0. The first two bytes in the sector must be either 0 or `'LK'`. The actual name of the AutoStart file is stored in the area allocated for the clipboard name. This area begins at offset 106. The first four bytes of the clipboard name field must contain the following value to indicate that the AutoStart file name follows: 0x006A 7068. After this 4-byte tag, 12 bytes of space remain, starting at offset 110. In these 12 bytes, the name of the AutoStart file is stored as a Pascal string, leaving up to 11 characters for the filename. The AutoStart file must reside in the root directory of the CD-ROM.

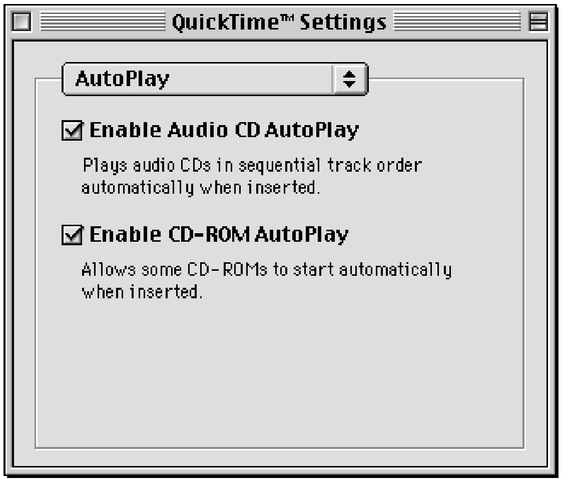
Because the AutoStart feature is based on the structure of an HFS disc, it is available only for the Macintosh. You can enable or disable AutoStart using the QuickTime Settings Control Panel.

AutoPlay for Audio CDs

QuickTime 2.5 allows you enable or disable the AutoPlay feature for audio CDs, and provides control over when you want a CD to play. If you want to use the Apple Audio CD Player or a similar application to control audio CD playback, launch that application before inserting the CD. If the Apple Audio CD Player or a similar application is not running, the CD begins playing from track 1 automatically when you insert the disc. Otherwise, you control when to start and stop the audio using your application software.

Figure 17-1 shows a screen shot of the AutoPlay control panel in QuickTime 3.

Figure 17-1 AutoPlay control panel


QuickTime™ Settings

AutoPlay

☑ Enable Audio CD AutoPlay

Plays audio CDs in sequential track order automatically when inserted.

☑ Enable CD-ROM AutoPlay

Allows some CD-ROMs to start automatically when inserted.

Media Keys

Media keys which authorize access to secured media files are explained in detail in Chapter 26, “Access Key Manager.”

Music Settings

For information about music settings, see *QuickTime Music Architecture*.

C H A P T E R 1 8

Video Output Components

This chapter discusses video output components. **Video output components** provide an interface for sending QuickTime video to devices that are not recognized as video displays by a computer's operating system. Any software that presents video data can be a client of a video output component.

About Video Output Components

QuickTime video output, which most often comes from QuickTime movies, can be displayed in windows that appear on a computer's desktop. Because these windows are created and managed by the computer's operating system, software that presents QuickTime video can use the operating system's video display services to specify which display (when there is more than one video display) and window to use for video output.

There are, however, many video output devices that are not recognized by operating systems. To display QuickTime video on these devices, your software can use video output components. The components, which are normally developed by the manufacturers of video output devices, provide a standard interface for video output to a device.

How Video Output Components Process Video Data

A video output component receives QuickTime video data and delivers data to a video output device for display. If the incoming data is in a format that the video output device can display directly, the video output component can simply send the data to the video output device. If the incoming data cannot be displayed directly, the video output component must use a transfer codec or decompressor component to convert the data to a format that the video output device can display.

If a video output device cannot directly display 32-bit RGB data or data in one of the other supported QuickTime pixel formats, the developers of the device are strongly encouraged to provide a transfer codec that accepts data in one of the supported QuickTime pixel formats (preferably 32-bit RGB) and converts it to data that can be displayed on the device. When this transfer codec is available, any QuickTime video can be displayed on the video output device: the Image Compression Manager can convert any QuickTime images to a supported QuickTime pixel format and then invoke the transfer codec to display the result.

If any special decompressors, such as a transfer codec, are needed for a video output device, the decompressors are included in the definitions of the component's display modes, as described in "Display Modes" (page 538). How transfer codecs work is described in "Accelerated Video Support" (page 225). How hardware developers can develop a transfer codec for their device is described in "Creating a Transfer Codec for a Video Output Component" (page 553).

Some video output devices do not accept pixels as input. For example, there are devices that display JPEG data directly. For these devices, a video output component can send the appropriate data directly, or it can invoke a compressor component to convert data in a pixel format to the appropriate data.

Display Modes

A video output device has one or more display modes. The characteristics of each mode determine how video is displayed. When any software displays video on a video output device, it must select which of the device's display modes to use.

The characteristics of a display mode include

- the height of the displayed image, in pixels
- the width of the displayed image, in pixels
- the horizontal resolution of the display, in pixels per inch
- the vertical resolution of the display, in pixels per inch
- the refresh rate of the display, in Hertz
- the pixel type of the display
- a text description of the display mode

The characteristics can also include a list of decompressor components required for the mode that are provided specifically for the video output device. If a video output device cannot directly display any of the pixel formats supported by QuickTime, the vendor of the device must provide one or more special decompressors to convert supported pixel formats to a format the device can display. If a video output device can display one or more of the pixel formats supported by QuickTime, the Image Compression Manager can use standard decompressors that are included with QuickTime, and the list of special decompressor components can be empty.

These characteristics, returned by the `QTVideoOutputGetDisplayModeList` function, are stored in a QT atom container. For a description of this QT atom container, see “Display Mode QT Atom Container” (page 555). For information about QT atom containers and how to traverse their contents, see “QuickTime Atoms” (page 47).

Using Video Output Components

This section explains how to use video output components in your software.

Video output components are standard components that are managed by the Component Manager. Their component type is `QTVideoOutputComponentType`. See the chapter “Component Manager” in *Mac OS For QuickTime Programmers* for more information about the Component Manager and about how to use components.

Apple has defined a functional interface for video output components. For each function of a video output component, there is a selector constant. These constants are listed in “Selectors for Video Output Component Functions” (page 555).

Selecting a Video Output Component

Listing 18-1 shows how to assemble a list of available video output components using the `FindNextComponent` function. This list can then be presented to the user.

The base video output component is a special component that provides services to other video output components. It is never connected to a display, and it has a component flag, `kQTVideoOutputDontDisplayToUser`, that indicates that it should not be included in a list of available video output components. The sample code shows how to check for this flag.

Listing 18-1 Displaying available video output components

```
ComponentDescription cd;
Component c = 0;

cd.componentType = QTVideoOutputComponentType;
cd.componentSubType = 0;
cd.componentManufacturer = 0;
cd.componentFlags = 0;
cd.componentFlagsMask = kQTVideoOutputDontDisplayToUser;

while (c = FindNextComponent (c, &cd)) {
    Handle nameHandle = NewHandle (0);

    GetComponentInfo (c, &cd, nameHandle, nil, nil);

    // add name to list

    DisposeHandle (nameHandle);
}
```

Choosing a Display Mode

After a video output component is chosen, the next step is to choose one of the component's available display modes. Your software does this by getting the QT atom container that contains descriptions of the available modes by calling the `QTVideoOutputGetDisplayModeList` function, traversing the atom container's contents using the `QTFindChildByIndex` function, examining the characteristics of each mode and setting aside any modes that are not appropriate for your software, and then optionally presenting a list of modes to the user to select from.

If your software does present a list of display modes to the user, it can obtain a string that describes each mode from the mode's `kQTVOName` atom with an ID of 1. The string doesn't include a leading length byte or a trailing null. Your software can determine the length of the string from the size of the atom.

When the mode has been chosen, your software calls the `QTVideoOutputSetDisplayMode` function to set the display mode.

Of display mode’s characteristics, the most important is whether the mode can display the video data. This is determined by the availability of a decompressor component that takes the video data as input and converts it to the type of data, specified by the `kQTVOPixelType` atom, required by the video output device. If a video output device can directly display one of the supported QuickTime pixel formats, the necessary decompressor component is included in QuickTime. If special decompressor components are required for the video output device, such as JPEG or other decompressors that deliver data directly to the video output hardware without creating a new pixel format, these decompressor components are described in `kQTVODecompressors` atoms.

Contents of the Display Mode QT Atom Container

To obtain descriptions of the display modes, your software must use the functions for traversing QT atom containers described in Chapter 1, “Movie Toolbox.”

At the root of the QT atom container returned by the `QTVideoOutputGetDisplayModeList` function are one or more atoms of type `kQTVODisplayModeItem`, each containing a definition of a display mode. Your software can traverse the display mode atoms by calling the `QTFindChildByIndex` function.

Within each `kQTVODisplayModeItem` atom are the following atoms:

- The atom of type `kQTVODimensions` with ID 1 contains two 32-bit integers. The first specifies the width, in pixels, of the display. The second specifies the height, in pixels, of the display.
- The atom of type `kQTVOResolution` with ID 1 contains two 32-bit fixed-point values. The first specifies the horizontal resolution of the display, in pixels per inch. The second specifies the vertical resolution of the display, in pixels per inch.

  By storing resolutions rather than an aspect ratio, QuickTime makes it easy for your software to compare values with values in QuickTime `ImageDescription` records. Your software can calculate the aspect ratio for the display mode by dividing the value for the horizontal resolution by the value for the vertical resolution.

- The atom of type `kQTVORefreshRate` with ID 1 contains a single 32-bit fixed-point value. This value specifies the refresh rate of the display in Hertz.
- The atom of type `kQTVOPixelType` with ID 1 contains a single 32-bit OSType value. This value specifies the type of pixel that is used by the display format:
  - Values of 1, 2, 4, 8, 16, 24 and 32 specify standard Mac OS RGB pixel formats with corresponding bit depths.
  - Values of 33, 34, 36 and 40 specify standard Mac OS gray-scale pixel formats with depths of 1, 2, 4, and 8 bits per pixel.
  - Other pixel formats are specified by four-character codes. There are currently codes for RGB pixel formats defined for Microsoft Windows and for several YUV formats. For information about pixel formats defined for Microsoft Windows, see *Introduction to QuickTime 3 for Windows Programmers.*
- The atom of type `kQTVOName` with ID 1 contains a string that describes the display mode. Your software can use this string when presenting a list of available display modes to the user. The string does not include a leading length byte or a trailing null. Your software can determine the length of the string by getting the size of the atom that contains it.
- Atoms of type `kQTVODecompressors` specify any special decompressors that are required for the video output device. If a video output device cannot directly display 32-bit RGB data or data in one of the other supported QuickTime pixel formats, a special decompressor is required to convert images to data that the video output device can display.

  Because `kQTVODecompressors` atoms are not required to have consecutive IDs, your software must use the `QTFindChildByIndex` function to iterate through the decompressors.

  Within each `kQTVODecompressors` atom are one or more atoms:
  - The atom of type `kQTVODecompressorType` with ID 1 contains an `OSType` value that specifies the type of compressed data that the decompressor can decompress. For example, a `kQTVODecompressorType` atom that contains `kMotionJPEGACodecType` can decompress Motion JPEG Format A data.
  - An atom of type `kQTVODecompressorComponent` with ID 1 is optional. If present, it contains a `DecompressorComponent` value that specifies a decompressor component that your software can use to decompress the data specified by the corresponding `kQTVODecompressorType` atom.

- □ An atom of type `kQTVODecompressorContinuous` with ID 1 is optional. If present, it contains a Boolean value that specifies whether the resulting video display will be continuous. If the value is `true`, data will be displayed without any visual gaps between successive images. If the value is `false`, data will be displayed, but there may be a visual gap (such as a black screen) between the display of images. If there is no `kQTVODecompressorContinuous` atom, your software should not make any assumptions about the performance of the decompressor.

Drawing to a Video Output

Your software draws images to a video output component in the same way that it draws to other video outputs: through a QuickDraw graphics world. You do this as follows:

1. After selecting the video output component and display mode, as described in the previous sections, call the `QTVideoOutputBegin` function to give your software exclusive use of the video output device.

   If any other software is using the video output device, `QTVideoOutBegin` returns the error code `videoOutputInUseErr`. If this happens, your software can call the `QTVideoOutGetCurrentClientName` function to get the name of the software and then display an alert that says that the output device is not available and notes the software that is currently using it.

2. Call `QTVideoOutputGetGWorld` to obtain the dimensions and resolution of the video output component's graphics world. This is the graphics world your software uses to draw images for the video.

3. Allocate a graphics world for your software with 32 bits per pixel and the same dimensions and resolution as the video output component's graphics world.

4. Draw the image to display into your software's graphics world.

5. Create an image sequence for transferring the image to the video output.

6. Call `DecompressSequenceFrameS` to send the image to the video output.

   This invokes the Image Compression Manager, which locates a image decompressor component that can convert the image into the pixel format required by the video output device.

The image decompressor component then delivers the video data to the video output device, which displays the image on the video output hardware.

7. Display another image or movie or call `QTVideoOutputEnd` to end the display and release control of the video output device.

Listing 18-2 illustrates how to create the image sequence and send it to the video output component.

Listing 18-2 Creating and sending an image sequence to a video output component

```
OSErr MakeImageSequenceForGWorld (GWorldPtr gw,
                                  GWorldPtr dest,
                                  long *imageSize,
                                  ImageSequence *seq)
{
    OSErr err = noErr;
    ImageDescriptionHandle desc = nil;
    ImageDescriptionPtr dp;
    PixMapHandle pixmap = gw->portPixMap;
    Rect bounds = (**pixmap).bounds;

    *seq = nil;
    *imageSize = ((**pixmap).rowBytes & 0x3fff) * dp->height;

    err = MakeImageDescriptionForPixMap (gw, &desc);
    if (err) goto bail;

    err = DecompressSequenceBeginS (seq, desc, dest, nil,
                                    &bounds, nil, ditherCopy,
                                    (RgnHandle)nil, 0,
                                    codecNormalQuality, anyCodec);
    if (err) goto bail;

bail:
    if (desc) DisposeHandle ((Handle)desc);
    return err;
}

{
    QTVideoOutputComponent videoOutput;
    GWorldPtr drawingBuffer = nil;
    GWorldPtr destGWorld;
    ImageSequence seq = nil;
    long imageSize;
    CodecFlags outFlags;

    QTVideoOutGetGWorld (videoOutput, &destGWorld);

    LockPixels (drawingBuffer->portPixMap);
    MakeImageSequenceForGWorld (drawingBuffer, destGWorld,
                               &imageSize, &seq);

    DecompressSequenceFrameS(seq,
                            GetPixBaseAddr (drawingBuffer->portPixMap),
                            imageSize, 0, &outFlags, nil);
}
```

Drawing to an Echo Port

Some video output devices can display video simultaneously on an external video display and in a window on a computer's desktop. To use this feature, your software draws to a graphics port for the window on the computer's desktop, known as the echo port, rather than the port that is normally used for the video output device. The video then appears on both displays, although in some cases the video on the desktop is displayed at a smaller size or lower frame rate.

To draw to both outputs at the same time, do the following:

1. Call the `ComponentFunctionImplemented` function to determine if the video output component supports the `QTVideoOutputSetEchoPort` function.
2. Call the `QTVideoOutputSetEchoPort` function to specify a window on the desktop in which to display video sent to the device.
3. Call the `SetMovieGWorld` function to specify the same window for the output of a movie.

This process is shown in Listing 18-3.

Listing 18-3 Drawing to an echo port

```
Movie               aMovie;
ComponentInstance   ci;
CGrafPtr            thePort;

/* instantiation of the video output component here */
/* creation of the graphics port here */

if (ComponentFunctionImplemented(ci, kQTVideoOutputSetEchoPortSelect)) {
    result = QTVideoOutputSetEchoPort(ci, thePort);
    SetMovieGWorld (aMovie, thePort, nil);
    StartMovie (aMovie);
}
```

The video then appears on both displays. Note that you bypass the graphics world that is normally used for the video output device; your software draws only to the echo port you specify with the `QTVideoOutputSetEchoPort` function.

Creating a Video Output Component

This section describes the routines a hardware developer must implement when creating a video output component.

The examples in this section show how your video output component can use the services of the base video output component provided by Apple Computer. If your component uses these services, you do not have to implement the entire API for a video output component. You simply implement the functions described here, and the base video output component handles the rest. For most of the functions, you extend functions already included in the base video output component, which is much faster than providing complete implementations of these functions yourself. If the base video output component's implementation of any of these functions returns an error, the function in your video output component must immediately return with the same error. If the base video output component's implementation completes successfully, then your video output component's function provides the remainder of the implementation.

Before reading this section, you should be familiar with how to create components. See “Component Manager” in *Mac OS For QuickTime Programmers* for a complete description of components, how to use them, and how to create them. For further information about using the Component Manager with QuickTime for Windows, see *Introduction to QuickTime 3 for Windows Programmers.*

Connecting to the Base Video Output Component

To use the services of the base video output component, your video output component must open a connection to the base video output component. It does this in its routine for processing open requests from the Component Manager. How to connect to the base video output component is shown in Listing 18-4.

Listing 18-4 Connecting to the base video output component

```
QTVideoOutputComponent baseVideoOutput;
OSErr err;

err = OpenADefaultComponent (kVideoOutputComponentType,
                             kVideoOutputComponentBaseSubType,
                             &baseVideoOutput);

err = ComponentSetTarget (baseVideoOutput,
                         self);

globals->baseVideoOutput = baseVideoOutput;
```

Providing a Display Mode List

Your video output component must implement its own `QTVideoOutputGetDisplayModeList` function. This function is required for all video output components.

Starting Video Output

Listing 18-5 shows how your video output component can start video output.

Listing 18-5 Starting video output

```
pascal ComponentResult MyQTVideoOutputBegin (Globals storage)
{
    ComponentResult err;
    long mode;

    // call the default implementation
    err = QTVideoOutputBegin (storage->baseVideoOutput);
    if (err) goto bail;

    // get the selected mode
    err = QTVideoOutputGetDisplayMode (storage->self, &mode);
    if (err) goto bail;

    // switch the hardware to the selected mode

    // remember that we now own the hardware
    storage->ownHardware = true;

bail:
    if ((err != noErr) && (storage->ownHardware == true))
        QTVideoOutputEnd (storage-> baseVideoOutput);
    return err;
}
```

The default implementation of the `QTVideoOutputBegin` function ensures that the hardware is not currently in use by other software. It also ensures that a valid display mode has been set with either the `QTVideoOutputSetDisplayMode` function or the `QTVideoRestoreSettings` function.

Ending Video Output

Listing 18-6 shows how your video output component can stop video output. The implementation of this function is similar to the implementation of `QTVideoOutputEnd`, but here the default implementation must be called after the hardware has been released.

Listing 18-6 Ending video output

```
pascal ComponentResult MyQTVideoOutputEnd (Globals storage)
{
    ComponentResult err;

    // check that Begin has been called
    if (storage->ownHardware == false) {
        err = paramErr;
        goto bail;
    }

    // release the hardware

    // call default implementation
    QTVideoOutputEnd (storage->baseVideoOutput);

    // remember that we no longer own the hardware
    store->ownHardware = false;

bail:
    return err;
}
```

In the implementation of `QTVideoOutputEnd`, your component should also display a default image on the video output device to indicate that the device is no longer in use by other software.

Implementing the QTVideoOutputSaveState Function

If your video output component uses any custom settings, your component must implement its own `QTVideoOutputSaveState` function to save them. If your video output component has no custom settings, it can use the default `QTVideoOutputSaveState` implementation provided by the base video output component. Listing 18-7 shows an implementation of the `QTVideoOutputSaveState` function that saves custom settings. The function creates a QT atom container for storing the settings.

Listing 18-7 Extending the `QTVideoOutputSaveState` function

```
pascal ComponentResult MyQTVideoOutputSaveState (Globals storage,
                                        QTAtomContainer *settings)
{
    OSErr err;

    // call default implementation
    err = QTVideoOutputSaveState (storage->baseVideoOutput, settings);
    if (err) goto bail;


    // add custom parameter(s)
    err = QTInsertChild (*settings,kParentAtomIsContainer,
                         'FOOB', 1, 0,
                         sizeof (storage->customSetting),
                         &storage->customSetting, nil);
    if (err) goto bail;

bail:
    return err;
}
```

Implementing the QTVideoOutputRestoreState Function

If your video output component saves custom settings with its own implementation of the `QTVideoOutputSaveState` function, it must also implement a `QTVideoOutputRestoreState` function to restore the settings. If your video output component has no custom settings, it can use the default `QTVideoOutputRestoreState` implementation provided by the base video output component. Listing 18-8 shows an implementation of the `QTVideoOutputRestoreState` function that restores custom settings from the QT atom container in which they are stored.

Listing 18-8 Restoring custom settings

```
pascal ComponentResult MyQTVideoOutputRestoreState (Globals storage,
                                        QTAtomContainer settings)
{
    OSErr err;
    QTAtom atom;

    // call default implementation
    err = QTVideoOutputRestoreState (storage->baseVideoOutput, settings);
    if (err) goto bail;

    // get custom parameter(s)
    atom = QTFindChildByID (settings, kParentAtomIsContainer, 'FOOB',
                            1, nil);
    if (atom != 0) {
        long dataSize;
        Ptr dataptr;

        QTGetAtomDataPtr (settings, atom, &dataSize, &dataPtr);
        storage->customSetting = *(SettingsType *)dataPtr;
    }
    else {
        // reset custom settings to default values
    }

bail:
    return err;
}
```

Implementing the QTVideoOutputGetGWorldParameters Function

Your video output component must also implement the `QTVideoOutputGetGWorldParameters`. This function is not called by applications or other clients of your component; it is called by the base video output component as part of the implementation of the `QTVideoOutputGetGWorld` function.

```
pascal ComponentResult QTVideoOutputGetGWorldParameters (
    QTVideoOutputComponent vo,
    Ptr *baseAddr,
    long *rowBytes,
    CTabHandle *colorTable);
```

In the `baseAddr` parameter, your video output component must return the address at which to display pixels. If your component does not display pixels, return 0 for this parameter.

In the `rowBytes` parameter, your video output component must return the width of each scan line in bytes. If your component does not display pixels, return the width of the current display mode.

In the `colorTable` parameter, your video output component must return the color table to be used. If your component does not use a color table, return `nil`.

Controlling Other Hardware Features

If the video output device includes features that can be controlled by any of the following functions, the video output component must implement the functions for those features.

```
QTVideoOutputGetIndSoundOutput
QTVideoOutputGetIndImageDecompressor
QTVideoOutputGetClock
QTVideoOutputCustomConfigureDisplay
QTVideoOutputSetEchoPort
```

Delegating Other Component Calls

Your video output component's dispatcher must delegate all component selectors it doesn't handle itself to the base video output component. It can do this by calling the `DelegateComponentCall` function.

Closing the Connection to the Base Video Output Component

When your video output component closes, it must close its connection to the base video output component by calling the `CloseComponent` function.

Creating a Transfer Codec for a Video Output Component

If you are the manufacturer of a video output device, you need to provide a video output component for your device as described in "Creating a Video Output Component" (page 546). In addition, if your video output device cannot display a pixel format defined by QuickTime, you should include a special decompressor, known as a transfer codec, that converts one of the supported QuickTime pixel formats (preferably 32-bit RGB) to data that the device can display. When this transfer codec is available, the QuickTime Image Compression Manager automatically uses it together with its built-in decompressors. This, in turn, lets applications or other software draw any QuickTime video directly to the video output component's graphics world. This section gives an overview of developing this transfer codec. For more information about transfer codecs and how to develop one, see "Accelerated Video Support" (page 225).

Video Output Components Reference

This section describes the constants, data structures, and functions that are specific to video output components.

Constants

This section provides details on component type, atom type, and function selector constants.

Component Instance, Type, and Subtype

```
typedef ComponentInstance QTVideoOutputComponent;
enum {
    QTVideoOutputComponentType = FOUR_CHAR_CODE('vout'),
    QTVideoOutputComponentBaseSubType = FOUR_CHAR_CODE('base')
};
```

Video Output Component Flag

The following flag indicates that a video output component is not connected to a display and should not be included in a list of components that are available to the user.

```
enum {
    kQTVideoOutputDontDisplayToUser = 1L << 0
};
```

Display Mode Atom Types

The following atom type constants specify atom types described in “Display Mode Atom Types” (page 554).

```
enum {
    kQTVODisplayModeItem = FOUR_CHAR_CODE('qdmi'),
    kQTVODimensions = FOUR_CHAR_CODE('dimn'),
        /* atom contains two longs - pixel count - width, height */
    kQTVOResolution = FOUR_CHAR_CODE('resl'),
        /* atom contains two Fixed - hRes, vRes in dpi */
    kQTVORefreshRate = FOUR_CHAR_CODE('refr'),
        /* atom contains one Fixed - refresh rate in Hz */
    kQTVOPixelType = FOUR_CHAR_CODE('pixl'),
        /* atom contains one OSType - pixel format of mode */
    kQTVOName = FOUR_CHAR_CODE('name'),
        /* atom contains string (no length byte) - name of mode for display to user */
    kQTVODecompressors = FOUR_CHAR_CODE('deco'),
        /* atom contains other atoms indicating supported decompressors */
        /* kQTVODecompressors sub-atoms */
    kQTVODecompressorType = FOUR_CHAR_CODE('dety'),
        /* atom contains one OSType - decompressor type code */
    kQTVODecompressorContinuous = FOUR_CHAR_CODE('cont'),
        /* atom contains one Boolean - true if this type is displayed continuously */
    kQTVODecompressorComponent = FOUR_CHAR_CODE('cmpt')
        /* atom contains one Component - component id of decompressor */
};
```

Selectors for Video Output Component Functions

The following constants are the selectors for functions of a video output component.

```
enum {
    kQTVideoOutputGetDisplayModeListSelect = 0x0001,
    kQTVideoOutputGetCurrentClientNameSelect = 0x0002,
    kQTVideoOutputSetClientNameSelect = 0x0003,
    kQTVideoOutputGetClientNameSelect = 0x0004,
    kQTVideoOutputBeginSelect = 0x0005,
    kQTVideoOutputEndSelect = 0x0006,
    kQTVideoOutputSetDisplayModeSelect = 0x0007,
    kQTVideoOutputGetDisplayModeSelect = 0x0008,
    kQTVideoOutputCustomConfigureDisplaySelect = 0x0009,
    kQTVideoOutputSaveStateSelect = 0x000A,
    kQTVideoOutputRestoreStateSelect = 0x000B,
    kQTVideoOutputGetGWorldSelect = 0x000C,
    kQTVideoOutputGetGWorldParametersSelect = 0x000D,
    kQTVideoOutputGetIndSoundOutputSelect = 0x000E,
    kQTVideoOutputGetClockSelect = 0x000F,
    kQTVideoOutputSetEchoPortSelect = 0x0010
};
```

Data Types

This section describes the QT atom container used to specify the display modes that are supported by a video display component.

Display Mode QT Atom Container

The `QTVideoOutputGetDisplayModeList` function returns a list of the display modes supported by a video display component. This list is contained in the QT atom container described in this section. For more information about QT atom containers, see "QuickTime Atoms" (page 47).

At the root of the QT atom container returned by the `QTVideoOutputGetDisplayModeList` function are one or more atoms of type `kQTVODisplayModeItem`, each containing a definition of a display mode. Your software can traverse the display mode atoms by calling the `QTFindChildByIndex` function.

Within each `kQTVODisplayModeItem` atom are the following atoms:

- The atom of type `kQTVODimensions` with ID 1 contains two 32-bit integers. The first specifies the width, in pixels, of the display. The second specifies the height, in pixels, of the display.
- The atom of type `kQTVOResolution` with ID 1 contains two 32-bit fixed-point values. The first specifies the horizontal resolution of the display, in pixels per inch. The second specifies the vertical resolution of the display, in pixels per inch.

  By storing resolutions rather than an aspect ratio, QuickTime makes it easy for your software to compare values with values in QuickTime `ImageDescription` records. Your software can calculate the aspect ratio for the display mode by dividing the value for the horizontal resolution by the value for the vertical resolution.
- The atom of type `kQTVORefreshRate` with ID 1 contains a single 32-bit fixed-point value. This value specifies the refresh rate of the display in Hertz.
- The atom of type `kQTVOPixelType` with ID 1 contains a single 32-bit OSType value. This value specifies the type of pixel that is used by the display format:
  - Values of 1, 2, 4, 8, 16, 24 and 32 specify standard Mac OS RGB pixel formats with corresponding bit depths.
  - Values of 33, 34, 36 and 40 specify standard Mac OS gray-scale pixel formats with depths of 1, 2, 4, and 8 bits per pixel.
  - Other pixel formats are specified by four-character codes. There are currently codes for RGB pixel formats defined for Microsoft Windows and for several YUV formats. For information about pixel formats defined for Microsoft Windows, see *Introduction to QuickTime 3 for Windows Programmers.*
- The atom of type `kQTVOName` with ID 1 contains a string that describes the display mode. Your software can use this string when presenting a list of available display modes to the user. The string does not include a leading length byte or a trailing null. Your software can determine the length of the string by getting the size of the atom that contains it.
- Atoms of type `kQTVODecompressors` specify any special decompressors that are required for the video output device. If a video output device cannot

directly display 32-bit RGB data or data in one of the other supported QuickTime pixel formats, a special decompressor is required to convert images to data that the video output device can display.

Because `kQTVODecompressors` atoms are not required to have consecutive IDs, your software must use the `QTFindChildByIndex` function to iterate through the decompressors.

Within each `kQTVODecompressors` atom are one or more atoms:

- □ The atom of type `kQTVODecompressorType` with ID 1 contains an `OSType` value that specifies the type of compressed data that the decompressor can decompress. For example, a `kQTVODecompressorType` atom that contains `kMotionJPEGACodecType` can decompress Motion JPEG Format A data.
- □ An atom of type `kQTVODecompressorComponent` with ID 1 is optional. If present, it contains a `DecompressorComponent` value that specifies a decompressor component that your software can use to decompress the data specified by the corresponding `kQTVODecompressorType` atom.
- □ An atom of type `kQTVODecompressorContinuous` with ID 1 is optional. If present, it contains a Boolean value that specifies whether the resulting video display will be continuous. If the value is `true`, data will be displayed without any visual gaps between successive images. If the value is `false`, data will be displayed, but there may be a visual gap (such as a black screen) between the display of images. If there is no `kQTVODecompressorContinuous` atom, your software should not make any assumptions about the performance of the decompressor.

Functions

This section has been divided into the following topics:

- "Controlling the Display Mode" describes functions for getting and setting the display mode used by a video output component.
- "Registering the Name of Your Software" discusses functions for registering and viewing the name of software using a video output component.
- "Controlling Video Output" describes functions for starting and stopping video output and for specifying the graphics world used for video output.
- "Finding Associated Components" discusses functions for finding clock and sound output components used with a video output component.

- "Saving and Restoring Component Configurations" describes functions for saving the current configuration of a video output component and later restoring the configuration.

Controlling the Display Mode

Each video output device has a finite number of display modes. Each mode has several characteristics, including width and height of the display, pixel depth, and video refresh rate. This section describes functions for getting and setting the display mode.

To get a list of the display modes supported by a video output component, call the `QTVideoOutputGetDisplayModeList` function. The list is a QT atom container, and list atoms contain the characteristics of each mode. You use QT atom container functions, such as `QTFindChildByIndex`, to extract the contents of the list.

To specify a display mode to use, call the `QTVideoOutputSetDisplayMode` function.

To find out the current display mode, call the `QTVideoOutputGetDisplayMode` function.

QTVideoOutputGetDisplayMode

The `QTVideoOutputGetDisplayMode` function gets the current display mode for a video output component.

```
pascal ComponentResult QTVideoOutputGetDisplayMode (
                     QTVideoOutputComponent vo,
                     long *displayModeID);
```

`vo`	Specifies the instance of a video output component. Your software obtains this reference when calling the Component Manager's `OpenComponent` or `OpenDefaultComponent` function.
`displayModeID`	Contains a pointer to the ID of the current display mode, or 0 if no display mode has been selected. The ID specifies a QT atom of type `kQTVODisplayModeItem` in the QT atom container returned

by the `QTVideoOutputGetDisplayModeList` function. For a description of the contents of this QT atom container, see "Display Mode QT Atom Container" (page 555).

DISCUSSION

If the `QTVideoOutputGetDisplayMode` function returns an atom ID of 0, it indicates that no display mode has been selected.

QTVideoOutputGetDisplayModeList

The `QTVideoOutputGetDisplayModeList` function gets a list of the display modes supported by a video output component.

```
pascal ComponentResult QTVideoOutputGetDisplayModeList (
                        QTVideoOutputComponent vo,
                        QTAtomContainer *outputs);
```

`vo`	Specifies the instance of a video output component. Your software obtains this reference when calling the Component Manager's `OpenComponent` or `OpenDefaultComponent` function.
`outputs`	Contains a pointer to the QT atom container that lists the video modes supported by this component. For a description of the contents of this QT atom container, see "Display Mode QT Atom Container" (page 555).

DISCUSSION

When your software calls `QTVideoOutputGetDisplayModeList`, it must dispose of the QT atom container returned by the function by calling `DisposeQTAtomContainer`.

QTVideoOutputSetDisplayMode

You can use the `QTVideoOutputSetDisplayMode` function in your software to specify the display mode to be used by a video output component.

```
pascal ComponentResult QTVideoOutputSetDisplayMode(
                         QTVideoOutputComponent vo,
                         long displayModeID);
```

`vo`	Specifies the instance of a video output component for the request. Your software obtains this reference when calling the Component Manager's `OpenComponent` or `OpenDefaultComponent` function.
`displayModeID`	The ID of the display mode to use. The ID specifies a QT atom of type `kQTVODisplayModeItem` in the QT atom container returned by the `QTVideoOutputGetDisplayModeList` function. For a description of the contents of this QT atom container, see "Display Mode QT Atom Container" (page 555).

DISCUSSION

When software changes the display mode with the `QTVideoOutputSetDisplayMode` function, the change does not take effect until the next time the software calls the `QTVideoOutputBegin` function for the video output component. This lets the software change other output settings before displaying the video.

Registering the Name of Your Software

After your software has established a connection to a video output component, you can register its name with the instance of that component by calling the `QTVideoOutputSetClientName` function. The name can then be used by `QTVideoOutputGetCurrentClientName` (page 565) to specify which software has exclusive access to the video output device controlled by the component.

Although several applications or other software can connect to a video output component at the same time, only one of them at a time can have access to the video output device controlled by the component. Use `QTVideoOutputBegin` (page 563) to gain exclusive access to the video output device and

QTVideoOutputEnd (page 565) to relinquish exclusive access when your software has finished using the device.

To get the name of the application or other software that is registered with an instance of a video output component, call QTVideoOutputGetClientName (page 561).

QTVideoOutputGetClientName

You can use the QTVideoOutputGetClientName function to get the name of the application or other software that is registered with an instance of a video output component.

```
pascal ComponentResult QTVideoOutputGetClientName (
                        QTVideoOutputComponent vo,
                        Str255 str);
```

vo	Specifies the instance of a video output component for the request. Your software obtains this reference when it calls the Component Manager's OpenComponent or OpenDefaultComponent function.
str	The name of the application or other software that is registered with the component instance.

QTVideoOutputSetClientName

You use the QTVideoOutputSetClientName function to register the name of an application or other software with an instance of a video output component.

```
pascal ComponentResult QTVideoOutputSetClientName (
                        QTVideoOutputComponent vo,
                        ConstStr255Param str);
```

`vo`	Specifies the instance of a video output component for the request. Your software obtains this reference when it calls the Component Manager's `OpenComponent` or `OpenDefaultComponent` function.
`str`	The name of the application or other software to be registered.

DISCUSSION

The name you specify with the `QTVideoOutputSetClientName` function can later be used by `QTVideoOutputGetCurrentClientName` (page 565) to specify which software has exclusive access to the video output device controlled by the component.

Controlling Video Output

Video output components provide functions for configuring the video display, for starting and stopping video output, and for specifying the graphics world used for the display. This section describes these functions.

To display a dialog box in which the user can specify video settings, call the `QTVideoOutputGetConfigureDisplay` function.

To get a pointer to the graphics world used by the video output component, call the `QTVideoOutputGetGWorld` function.

To obtain exclusive access to the video hardware controlled by a video output component, call the `QTVideoOutputBegin` function.

To release access to the video hardware controlled by a video output component, call the `QTVideoOutputEnd` function.

To get the name of the software, if any, that has exclusive access to the video hardware controlled by a video output component, call the `QTVideoOutputGetCurrentClientName` function.

If a video output device can display video both on an external video display and in a window on a computer's desktop, you can use the `QTVideoOutputSetEchoPort` function to specify a window on the desktop in which to display video sent to the device.

QTVideoOutputBegin

You use the `QTVideoOutputBegin` function in your software to obtain exclusive access to the video hardware controlled by a video output component.

```
pascal ComponentResult QTVideoOutputBegin (QTVideoOutputComponent vo);
```

`vo`	Specifies the instance of a video output component. Your software obtains this reference when calling the Component Manager's `OpenComponent` or `OpenDefaultComponent` function.

DISCUSSION

When your software calls `QTVideoOutputBegin`, the video output component does the following:

- Acquires exclusive access to the video hardware controlled by the specified video output component or returns the `videoOutputInUseErr` result code if the video hardware is currently in use.

If the video hardware is available, the video output component also

- Enables the display mode last set with the `QTVideoOutputSetDisplayMode` function.
- Enables the video settings, if any, that were most recently specified by the user in a custom video configuration dialog box.

If the video output component supports `QTVideoOutputCustomConfigureDisplay` (page 564), your software can call the function to display a custom video configuration dialog box.

When your software no longer needs the video output component, release it by calling the `QTVideoOutputEnd` function.

If the `QTVideoOutputBegin` function returns the `videoOutputInUseErr` result code that indicates that the video hardware is currently in use, your software can get the name of the application or other software that is using the hardware by calling `QTVideoOutputGetCurrentClientName` (page 565). You can then display an alert to the user that

- says that the video hardware is in use
- specifies the name of the software using the video hardware.

If your software needs to change the display mode, it must change it before calling `QTVideoOutputBegin`. It cannot change the display mode between calls to `QTVideoOutputBegin` and `QTVideoOutputEnd`.

RESULT CODE

`videoOutputInUseErr`	–2090	Another application or other software has exclusive use of the video output device controlled by this component.

QTVideoOutputCustomConfigureDisplay

If a video output component supports the optional `QTVideoOutputCustomConfigureDisplay` function, your software can call the function to display a custom video configuration dialog box. This dialog box can include settings that are specific to the video device controlled by the video output component.

```
pascal ComponentResult QTVideoOutputCustomConfigureDisplay (
                        QTVideoOutputComponent vo,
                        ModalFilterUPP filter);
```

`vo`	Specifies the instance of a video output component for this request. Your software obtains this reference when calling the Component Manager’s `OpenComponent` or `OpenDefaultComponent` function.
`filter`	Specifies a Dialog Manager filter procedure for the video output component to use for the dialog box. The filter allows the software to process events while the dialog box is displayed.

DISCUSSION

Your software can determine if a video output component supports the `QTVideoOutputCustomConfigureDisplay` function by calling the `ComponentFunctionImplemented` function for the component with the routine selector `kQTVideoOutputCustomConfigureDisplaySelect`.

RESULT CODES

`userCanceledError`	–128	The user clicked Cancel in the dialog box.
`badComponentSelector`	$800008002	The video output component does not provide a configuration dialog.

QTVideoOutputEnd

You use the `QTVideoOutputEnd` function in your software to release access to the video hardware controlled by a video output component.

```
pascal ComponentResult QTVideoOutputEnd (QTVideoOutputComponent vo);
```

`vo`	Specifies the instance of a video output component. Your software obtains this reference when calling the Component Manager's `OpenComponent` or `OpenDefaultComponent` function.

DISCUSSION

Your software should release access to a video output component as soon as it is done using the video hardware controlled by the component.

If you close the instance of a video output component that currently has exclusive access to video hardware, the video output component automatically calls `QTVideoOutputEnd` to release the hardware.

QTVideoOutputGetCurrentClientName

You can use the `QTVideoOutputGetCurrentClientName` function in your software to get the name of the software, if any, that has exclusive access to the video hardware controlled by a video output component.

```
pascal ComponentResult QTVideoOutputGetCurrentClientName
                        (QTVideoOutputComponent vo, Str255 str);
```

`vo`	Specifies the instance of a video output component for this request. Your software obtains this reference when calling the Component Manager's `OpenComponent` or `OpenDefaultComponent` function.
`str`	The name of the software that has exclusive access to the video hardware controlled by a video output component, or a zero-length string if no software currently has access.

DISCUSSION

If video hardware is unavailable because other software is using it, your software can inform users by getting the name of the software with the `QTVideoOutputGetCurrentClientName` function and displaying the name in an alert box.

QTVideoOutputGetGWorld

You use the `QTVideoOutputGetGWorld` function in your software to get a pointer to the graphics world used by a video output component.

```
pascal ComponentResult QTVideoOutputGetGWorld (
                        QTVideoOutputComponent vo,
                        GWorldPtr *gw);
```

`vo`	Specifies the instance of a video output component for this request. Your software obtains this reference when calling the Component Manager's `OpenComponent` or `OpenDefaultComponent` function.
`gw`	Contains a pointer to the graphics world used by the video output component to display images.

DISCUSSION

If the pixel format of the graphics world is 1, 2, 4, 8, 16, or 32, your software can use either QuickDraw or QuickTime to draw graphics to it. If the graphics world has any other pixel format, your software must use QuickTime functions draw to it.

Your software can pass the pointer returned by the `QTVideoOutputGetGWorld` function to the `SetMovieGWorld`, `DecompressSequenceBegin`, `DecompressSequenceBeginS`, `DecompressImage`, and `FDecompressImage` functions or, for Microsoft Windows, the equivalent functions described in *Introduction to QuickTime 3 for Windows Programmers.*

Your software can call `QTVideoOutputGetGWorld` only between calls to `QTVideoOutputBegin` and `QTVideoOutputEnd`. When your software calls `QTVideoOutputEnd`, the video output component automatically disposes of the graphics world. If your software needs to use the graphics world after calling `QTVideoOutputEnd`, it must call `QTVideoOutputGetGWorld` function again after the next time it calls `QTVideoOutputBegin`.

IMPORTANT

Your software must not call `DisposeGWorld` to dispose of the graphics world used by a video output component. The video output component automatically disposes of the graphics world when your software calls `QTVideoOutputEnd`.

QTVideoOutputGetGWorldParameters

A video output components's `QTVideoOutputGetGWorldParameters` function is called by the base video output component as part of its implementation of the `QTVideoOutputGetGWorld` function. `QTVideoOutputGetGWorldParameters` is not called by applications or other client software.

```
pascal ComponentResult QTVideoOutputGetGWorldParameters (
                    QTVideoOutputComponent vo,
                    Ptr *baseAddr,
                    long *rowBytes,
                    CTabHandle *colorTable);
```

`vo`	Specifies an instance of your video output component.
`baseAddr`	Contains the address at which to display pixels. If your video output component does not display pixels, return 0 for this parameter.

`rowBytes`	Specifies the width of each scan line in bytes. If your video output component does not display pixels, return the width of the current display mode.
`CTabHandle`	Identifies the color table to be used. If your video output component does not use a color table, return `nil`.

QTVideoOutputSetEchoPort

If a video output device can display video both on an external video display and in a window on a computer's desktop, you can use the `QTVideoOutputSetEchoPort` function to specify a window on the desktop in which to display video sent to the device. When your software sends video to the window you specify, the video is both displayed in the window and sent to the normal output of the video output device.

```
pascal ComponentResult QTVideoOutputSetEchoPort (
                        QTVideoOutputComponent vo,
                        CGrafPtr echoPort);
```

`vo`	Specifies the instance of a video output component for this request. Your software obtains this reference when calling the Component Manager's `OpenComponent` or `OpenDefaultComponent` function.
`echoPort`	Specifies the window on the computer's desktop in which to display the video.

DISCUSSION

When an output device can display video both on an external video display and in a window on a computer's desktop, the video displayed on the desktop is often at a smaller size and/or lower frame rate.

Finding Associated Components

Video output components provide functions for finding other components associated with them. This section describes these functions.

To find any sound output components associated with a video output component, call the `QTVideoOutputGetIndSoundOutput` function.

To find a clock component associated with a video output component, call the `QTVideoOutputGetClock` function.

QTVideoOutputGetClock

If a video output component supports the optional `QTVideoOutputGetClock` function, your software can call the function to get a pointer to the clock component associated with the video output component.

```
pascal ComponentResult QTVideoOutputGetClock (
                        QTVideoOutputComponent vo,
                        ComponentInstance *clock);
```

`vo`	Specifies the instance of a video output component for this request. Your software obtains this reference when calling the Component Manager's `OpenComponent` or `OpenDefaultComponent` function.
`clock`	Contains a pointer to the clock component associated with the video output component.

DISCUSSION

Your software can use the clock component returned by the `QTVideoOutputGetClock` function to synchronize video and sound for a movie to the rate of the display. To associate the instance of the clock component with a movie, call the `SetMovieMasterClock` function.

Because a change to the display mode could affect a clock component, your software should call `QTVideoOutputGetClock` only between calls to `QTVideoOutputBegin` and `QTVideoOutputEnd`, when it is not possible to change the display mode.

When your software calls the `QTVideoOutputEnd` function, the video output component disposes of the instance of the clock component returned by the `QTVideoOutputGetClock` function. Because of this, software that uses the clock to control a movie must reset the clock for the movie to the default clock (by calling `SetMovieMasterClock` with `nil` as the value of the clock component) before calling `QTVideoOutputEnd`.

RESULT CODES

`badComponentSelector`	$800008002	The video output component does not have an associated clock component.

QTVideoOutputGetIndSoundOutput

If a video output component supports the optional `QTVideoOutputGetIndSoundOutput` function, your software can call the function to determine which sound output components are associated with the video output component.

```
pascal ComponentResult QTVideoOutputGetIndSoundOutput (
                        QTVideoOutputComponent vo,
                        long index,
                        Component *outputComponent);
```

`vo`	Specifies the instance of a video output component for this request. Your software obtains this reference when calling the Component Manager's `OpenComponent` or `OpenDefaultComponent` function.
`index`	Specifies which of the sound output components to return. The index of the first component is 1.
`outputComponent`	Contains a pointer to a sound output component associated with the video output component that is specified by the `index` parameter.

DISCUSSION

Your software can display sound output components returned by the `QTVideoOutputGetIndSoundOutput` function in a dialog box and let the user choose which outputs to use for movie playback.

Saving and Restoring Component Configurations

Video output components provide functions for saving the current configuration of a video output component and later restoring the configuration. This section describes these functions.

To save the current configuration of a video output component, call the `QTVideoOutputSaveState` function.

To restore a previously saved configuration of a video output component, call the `QTVideoOutputRestoreState` function.

QTVideoOutputRestoreState

You use the `QTVideoOutputRestoreState` function in your software to restore the previously saved state of a video output component.

```
pascal ComponentResult QTVideoOutputRestoreState (
                        QTVideoOutputComponent vo,
                        QTAtomContainer state);
```

`vo`	Specifies the instance of a video output component for this request. Your software obtains this reference when calling the Component Manager's `OpenComponent` or `OpenDefaultComponent` function.
`state`	A QT atom container, returned earlier by the `QTVideoOutputSaveState` function, that contains state information for the video output component.

DISCUSSION

If your software saves state information to disk, it must read the QT atom container structure from disk before calling `QTVideoOutputRestoreState`.

When your software restores state information for a video output component, the current display mode may change. Because of this, your software must call `QTVideoOutputRestoreState` before calling `QTVideoOutputStart`.

QTVideoOutputSaveState

You use the `QTVideoOutputSaveState` function to save state information for an instance of a video output component.

```
pascal ComponentResult QTVideoOutputSaveState (
                      QTVideoOutputComponent vo,
                      QTAtomContainer *state);
```

`vo`	Specifies the instance of a video output component for this request. Your software obtains this reference when calling the Component Manager's `OpenComponent` or `OpenDefaultComponent` function.
`state`	Contains a pointer to complete information about the video output component's current configuration.

DISCUSSION

When your software saves state information for an instance of a video output component, it can restore this information when reconnecting to the component by calling the `QTVideoOutputRestoreState` function.

When your software calls `QTVideoOutputSaveState`, it must dispose of the QT atom container returned by the function by calling `DisposeQTAtomContainer`.

QuickTime Audio

This chapter discusses new features of QuickTime audio. For information about audio support and the uses of the Sound Manager, refer to Chapter 2 of *Inside Macintosh: Sound*.

This chapter focuses on issues that are relevant to QuickTime. Note that Sound Manager sources are now multi-platform, so that the same manager is available on all platforms where QuickTime is available, and contains the same API and features.

This chapter also discusses how QuickTime 3 handles compressed audio, in addition to the two recent extensions defined for the `SoundDescription` sample description record. The first extension is the addition of `slope`, `intercept`, `minClip`, and `maxClip` parameters for audio. The second is the ability to store data specific to a given audio decompressor in the `SoundDescription` record.

New Features of QuickTime Audio

QuickTime 3 automatically installs the latest version of the Sound Manager.

Multi-platform Support

The Sound Manager sources are now multi-platform, i.e., built for the Mac OS, Windows 95 and NT, and any other platform that supports QuickTime. This means the same Sound Manager software is available for each platform, containing the same API and features.

Dealing with the Issue of “Endianness”

Multi-platform support raises the issue of “endianness.” Basically, endian conversion is treated as a compression conversion. Any non-native endian format is required to be “decompressed” into the native format.

How QuickTime 3 Handles Compressed Audio

QuickTime 3 defines version 1 of the `SoundDescription` sample description record. Note that for purposes of this discussion, a QuickTime sound sample description chunk describes the format of a collection of audio samples.

The existing description is shown in Listing 19-1 for reference.

Listing 19-1 The original SoundDescription sample description

```
struct SoundDescription {
    long        descSize;       /* total size of SoundDescription
                                   including extra data */
    long        dataFormat;     /* sound format */
    long        resvd1;         /* reserved for apple use. set to zero */
    short       resvd2;         /* reserved for apple use. set to zero */
    short       dataRefIndex;
    short       version;        /* which version is this data */
    short       revlevel;       /* what version of that codec did this */
    long        vendor;         /* whose  codec compressed this data */
    short       numChannels;    /* number of channels of sound */
    short       sampleSize;     /* number of bits per sample */
    short       compressionID;  /* unused. set to zero. */
    short       packetSize;     /* unused. set to zero. */
    UnsignedFixed sampleRate;   /* sample rate sound is captured at */
};
typedef struct SoundDescription SoundDescription;
```

Version 1 of this record includes four extra fields to store information about compression ratios. It also defines how other extensions are added to the `SoundDescription` record.

```
struct SoundDescriptionV1 {
    // original fields
    SoundDescription    desc;
    // fixed compression ratio information
    unsigned long   samplesPerPacket;
    unsigned long   bytesPerPacket;
    unsigned long   bytesPerFrame;
    unsigned long   bytesPerSample;
    // additional atom-based fields --
    // ([long size, long type, some data], repeat)
};
```

The version 1 sound description is a superset of the version 0 sound description. The new fields are taken directly from the `CompressionInfo` structure currently used by the Sound Manager to describe the compression ratio of fixed ratio audio compression algorithms. They are described in detail in *Inside Macintosh: Sound*. If these fields are not used, they are set to 0. File readers only need to check to see if `samplesPerPacket` is 0. The fields have been added to support compression algorithms which can be run at different compression ratios and to support more generic parsing of QuickTime sound tracks

IMPORTANT

It is necessary to know the compression ratio to rechunk or flatten the media. In the past, the only way to know the compression ratio was to directly query the audio decompressor. If this process was running on a computer without the decompressor (such as a server), it would not have enough information to correctly rechunk the audio. ▲

All other additions to the `SoundDescription` record are made using QT atoms. That means one or more atoms can be appended to the end of the SoundDescription record using the standard [size, type] mechanism used throughout the QuickTime movie resource architecture.

Extensions to the SoundDescription Record

Two extensions are defined to the `SoundDescription` record. The first is the `slope`, `intercept`, `minClip`, and `maxClip` parameters for audio as defined in Appendix D. This is represented as an atom of type `siSlopeAndIntercept`. The contents of the atom are:

```
struct SoundSlopeAndInterceptRecord {
    Float64                 slope;
    Float64                 intercept;
    Float64                 minClip;
    Float64                 maxClip;
};
typedef struct SoundSlopeAndInterceptRecord SoundSlopeAndInterceptRecord;
```

The second extension is the ability to store data specific to a given audio decompressor in the `SoundDescription` record. Some audio decompression algorithms, such as Microsoft's ADPCM, require a set of out-of-stream values to configure the decompressor. These are stored in a `siDecompressorSettings`. The contents of the `siDecompressorSettings` atom are dependent on the audio decompressor. If the QuickTime movie was created from a WAVE (`.WAV`) or AVI (`.avi`) file, the `siDecompressorSettings` atom is automatically created and set to the contents of the `WAVEFORMATEX` structure from that file. In this case, the `siDecompressorSettings` atom contains little-endian data.

At runtime, the contents of the type `siSlopeAndIntercept` and `siDecompressorSettings` atoms are provided to the decompressor component through the standard `SetInfo` mechanism of the Sound Manager. The `samplesPerPacket`, `bytesPerPacket`, `bytesPerFrame`, and `bytesPerSample` fields are also passed to the decompressor component via `SetInfo` in a `CompressionInfo` structure with the `siCompressionFactor` selector.

Constants for Additional Audio Compression Formats

QuickTime 3 defines constants for several additional audio compression formats. These include `kFloat32Format` and `kFloat64Format` for single and double precision (i.e., big endian IEEE) floating-point audio. It also defines `'alaw'` for aLaw audio. All DV audio from NTSC (format 60) DVC streams is `'dvca'`, regardless of the format of the data in the frame. In addition, a standard mapping for audio formats present in the Windows Audio Compression Manager is defined.

All ACM audio formats are defined with a 16-bit integer. These are mapped into a QuickTime four-character code by putting 'ms' in the first two characters and the ACM 16-bit value in the second two characters. So the Microsoft ADPCM algorithm (ACM value of 1) has a QuickTime audio compression code of (('MS' << 16) | 1). This enables standard mapping of ACM audio into the QuickTime movie format.

QuickTime 3 has built-in support to decompress the following additional audio formats:

- single-precision floating point
- double-precision floating point
- Microsoft ADPCM (ACM code 1)
- Intel / DVI IMA (ACM code 7), DV
- aLaw

Both floating-point decompressors support the type `siSlopeAndIntercept` atom to provide scaling and DC-offset support. QuickTime 3 has the ability to compress the following additional audio formats:

- single-precision floating point
- double-precision floating point
- aLaw

QuickTime 3 correctly imports compressed audio from AVI (`.avi`) and WAVE (`.WAV`) files. The AU file importer has also been enhanced to import aLaw, 8- and 16-bit uncompressed, and single- and double-precision floating point audio in addition to the uLaw that it handled previously. A Sound Designer II file importer has been added. An AU file exporter has been added which can generate aLaw, uLaw, single and double precision floating point, and uncompressed AU files. (Note that most other readers can only handle uLaw, however).

DV Audio Decompressor Component

The Sound Manager includes a DV audio decompressor component. This component, which works with Sound Manager version 3.3 or later, can decompress DV audio in any of the formats supported by DV sources. These formats are:

- 2 audio channels, 12-bit encoding, 32K samples per second
- 2 audio channels, 12-bit encoding, 44.1K samples per second
- 2 audio channels, 12-bit encoding, 48K samples per second
- 2 audio channels, 16-bit encoding, 32K samples per second
- 2 audio channels, 16-bit encoding, 44.1K samples per second

- 2 audio channels, 16-bit encoding, 48K samples per second
- 4 audio channels, 12-bit encoding, 32K samples per second

The file type for all DV audio formats is `kDVAudioFormat`.

Note that the DV audio decompressor component provides the ability to decode audio data stored in a DVC stream.

Using QuickTime Audio

Creating a 16-bit, 22K Uncompressed WAVE File Using QuickTime 3

The following is an example of how you can convert an 8-bit, 22K .wav file to a 16-bit, 22K IMAPCM .wav file using QuickTime 3.

While QuickTime can create WAVE files, it does not support creating IMA-compressed WAVE files. It can play back WAVE files that contain IMA-compressed audio.

To create a 16-bit 22k uncompressed WAVE file using QuickTime 3, you perform the following steps:

4. Open an exporter.

```
ComponentInstance ci;
ci = OpenDefaultComponent(MovieExportType, kQTFileTypeWave);
```

5. Create a sound description for the audio format you want. If there are values you don’t care about, you can leave them unspecified. In this example, the number of channels is not indicated. The exporter will base the channel count on the source movie.

```
SoundDescriptionHandle desc;

desc = (SoundDescriptionHandle )NewHandleClear(sizeof(SoundDescripion));
(**desc).descSize = sizeof(SoundDescription);
```

```
(**desc).sampleSize = 16;
(**desc).sampleRate = 22050L << 16;
(**desc).dataFormat = k16BitLittleEndianFormat;
```

6. Specify the export component in which format you want the audio.

```
MovieExportSetSampleDescription(ci, (SampleDescriptionHandle)desc,
SoundMediaType);
```

7. Perform the export operation.

```
ConvertMovieToFile(theMovie, nil, &outputFile, kQTFileTypeWave,
                    OSTypeConst('TVOD'), -1, nil, 0, ci);
```

8. After you have finished, dispose everything that you have created.

```
CloseComponent(ci);
DisposeHandle((Handle)desc);
```

Standard Sound Dialog Component

This chapter describes the standard sound dialog component, a QuickTime component with an API for configuring sound settings.

New Features of Standard Sound Dialog Components

Standard Compression Components and Settings

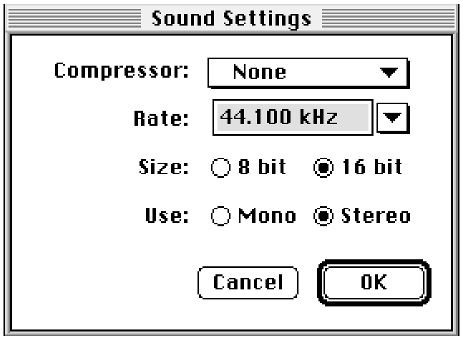
QuickTime 3 adds two settings-related component calls to the standard compression components for both the video and sound. The `SCGetSettingsAsAtomContainer` returns a QT atom container with the current configuration of the particular compression component. `SCSetSettingsFromAtomContainer` resets the compression component's current configuration. Applications that want to save settings for standard compression components should use these new calls.

These calls are discussed in "Standard Sound Dialog Component Reference" (page 584).

About the Standard Sound Dialog Component

The standard sound dialog component allows your application to present the user with a dialog box to configure sound settings and to get the settings that the user provides. The standard sound dialog component API is based on the standard image-compression dialog component API, which offers similar programmatic controls for image compression settings. An example of the dialog box provided by the standard sound dialog component is shown in Figure 20-1.

Figure 20-1 Standard Sound Dialog



Sound Settings

Compressor: None

Rate: 44.100 kHz

Size: 8 bit [✓ 16 bit]

Use: Mono [✓ Stereo]

The standard sound dialog component implements calls that are appropriate for sound processing. Because the standard sound dialog component is based on the standard compression component, many calls were originally developed for image-related settings. The names of some calls may therefore seem counterintuitive when used in the context of sound compression.

The standard sound dialog component is currently used for the configuration dialog boxes in the sound and music movie exporter components.

Using the Standard Sound Dialog Component

To work with the standard sound dialog component, it is necessary to open a connection to the component using the Component Manager. An example of how to do this is shown in Listing 20-1.

Listing 20-1 Opening a connection to the standard sound dialog component

```
ComponentInstance ci;
ci = OpenDefaultComponent(StandardCompressionType,
        StandardCompressionSubTypeSound);
```

After obtaining a connection to the standard sound dialog component, it may be appropriate to configure the dialog box to present a reasonable set of values. This is done with `SCSetInfo` as shown in Listing 20-2. This example sets the sample rate to 32000 Hz, the sample size to 8 bits, the channel count to 1, and the compression format to MACE 6:1. If `SCSetInfo` is not called for a given parameter, that parameter defaults to an appropriate value.

Listing 20-2 Setting initial values for the dialog box

```
UnsignedFixed   rate;
short           sSize, cCount;
OSType          compType;
rate = FixRatio(32000, 1);
SCSetInfo(ci, scSoundSampleRateType, &rate);
sSize = 8;
SCSetInfo(ci, scSoundSampleSizeType, &sSize);
cCount = 1;
SCSetInfo(ci, scSoundChannelCountType, &cCount);
compType = kMACE6Compression;
SCSetInfo(ci, scSoundCompressionType, &compType);
```

It is sometimes necessary to restrict the list of compression types appearing in the compression menu. For example, some clients may not support generation of compressed data. The standard sound dialog component allows the client to provide a list of compression types that should be displayed. In the example in Listing 20-3, all compression types except for uncompressed are eliminated from the list.

Listing 20-3 Restricting the list of compression types

```
Handle compressionTypeList;
compressionTypeList = NewHandle(sizeof(OSType));
**(OSType **)compressionTypeList = kRawCodecType;
SCSetInfo(ci, scCompressionListType, &compressionTypeList);
```

To display the dialog box, use `SCRequestImageSettings` as shown in Listing 20-4. If the user cancels the dialog box, `userCanceledErr` is returned.

Listing 20-4 Displaying the dialog box

```
OSErr err;
err = SCRequestImageSettings(ci);
```

After the dialog box has been displayed, the settings can be retrieved using the `SCGetInfo` call with the appropriate selectors. The example in Listing 20-5 shows how to retrieve the selected sample rate, sample size, compression format, and number of channels.

Listing 20-5 Retrieving settings from the dialog box

```
UnsignedFixed       rate;
short               sSize, cCount;
OSType              compType;

SCGetInfo(ci, scSoundSampleRateType, &rate);
SCGetInfo(ci, scSoundSampleSizeType, &sSize);
SCGetInfo(ci, scSoundChannelCountType, &cCount);
SCGetInfo(ci, scSoundCompressionType, &compType);
```

It is also possible to retrieve all of the current settings in a single handle, as shown in the next example. This can be convenient when saving user settings:

```
Handle h;
SCGetInfo(ci, scSettingsStateType, &h);
```

Once the settings have been retrieved in the handle, they can be restored using `SCSetInfo`, as follows:

```
SCSetInfo(ci, scSettingsStateType, &h);
```

When you are finished with the standard sound dialog component, close the connection to the component as follows:

```
CloseComponent(ci);
```

Standard Sound Dialog Component Reference

This section describes new constants and functions of standard sound dialog components.

Constants

The following constants have been added to the list of selectors that can be passed to `SCGetInfo` and `SCSetInfo`. These selectors are only supported by the standard sound dialog component, not the standard image-compression dialog component.

`scSoundSampleRateType`
: A pointer to an `UnsignedFixed` value that represents the current sample rate.

`scSoundSampleSizeType`
: A pointer to a short integer that represents the current sample size. This value is either 8 or 16.

`scSoundChannelCountType`
: A pointer to a short integer that represents the current number of channels. This value is either 1 or 2.

`scSoundCompressionType`
: A pointer to a value of type `OSType` that represents the current compression type. This value is either `kRawCodecType` or one of the available sound compression formats.

`scCompressionListType`
: A pointer to a handle containing an array of values of type `OSType` that indicate the sound compression formats that may be presented to the user. Pass `nil` to `SCSetInfo` to reset the list to all available sound compression formats.

The standard sound dialog component also supports the following selectors.

`scPreferenceFlagsType`
: The only flag that is supported in the preferences is `scUseMovableModal`. All other flags should be set to zero. The preference flags are initialized to `scUseMovableModal` when the standard sound dialog component is instantiated.

`scExtendedProcsType`
: The only fields supported are `filterProc` and `refcon`. All other fields should be initialized to zero. The filter procedure should only be used to update background windows. It should not be used to intercept user interactions in the dialog box window itself.

`scSettingsStateType`
: This selector is supported in the same way as in the standard image-compression component.

Functions

The standard sound dialog component implements the following functions from the standard compression component interface. For more details, see the "Standard Image-Compression Dialog Components" chapter of *Inside Macintosh: QuickTime Components*.

`SCGetInfo`
: This function supports all of the selectors documented in the "Constants" section. If an unsupported selector is used, it returns `scTypeNotFoundErr`.

`SCSetInfo`
: This function supports all of the selectors documented in the "Constants" section. If an unsupported selector is used, it returns `scTypeNotFoundErr`.

`SCRequestImageSettings`	This function displays the sound settings dialog. If the user cancels the dialog box, the `userCanceledErr` is returned. To provide a filter procedure to process events while the dialog box is displayed, use `SCSetInfo` with the `scExtendedProcsType` selector.

SCGetSettingsAsAtomContainer

The `SCGetSettingsAsAtomContainer` routine retrieves the current configuration from the standard image-compression component.

```
pascal ComponentResult SCGetSettingsAsAtomContainer (
                        ComponentInstance  ci,
                        QTAtomContainer *settings);
```

`ci`	The standard compression component instance.
`settings`	The address where the newly-created atom container should be stored by the call.

DISCUSSION

The caller is responsible for disposing of the returned QT atom container.

SCSetSettingsFromAtomContainer

The `SCSetSettingsFromAtomContainer` routine sets the standard image-compression component's current configuration from the passed settings data.

```
pascal ComponentResult SCSetSettingsFromAtomContainer (
                        ComponentInstance  ci,
                        QTAtomContainer settings);
```

`ci`	Standard compression component instance.

`settings`	Contains a QT atom container reference to the settings.

DISCUSSION

The `settings` QT atom container may contain atoms other than those expected by the particular component type or may be missing certain atoms. The function will only use settings it understands.

The Base Image Decompressor

This chapter describes the **base image decompressor,** a component that performs tasks for image decompressor components, including the scheduling of asynchronous decompression operations.

You need the information in this chapter only if you are implementing an image decompressor component. For information on implementing an image decompressor component, see Chapter 4, "Image Compressor Components," in this reference and Chapter 4, "Image Compressor Components," in *Inside Macintosh: QuickTime Components*.

About the Base Image Decompressor

Whenever possible, an image decompressor component should handle asynchronous requests for decompression, as described in "Asynchronous Decompression" (page 223). If you are implementing an image decompressor component, you can include this capability with a minimum of additional programming by using the services of the base image decompressor. The base image decompressor handles the necessary scheduling, which frees you to concentrate on the details of decompression.

When you use the base image decompressor with an image decompressor component, your component must support functions that are called by the base image decompressor when necessary. Your component can then delegate a number of other function calls to the base image decompressor, which greatly simplifies the implementation of your component.

Using the Base Image Decompressor

To use the services of the base image decompressor, your image decompressor component must support functions that the base image decompressor calls when necessary. The following sections explain when the base image decompressor calls these functions and how your image decompressor component must respond. These sections also list standard image decompressor calls that your image decompressor must handle itself rather than delegate.

The base image decompressor and image decompressor components are managed by the Component Manager. See the “Component Manager” chapter in *Mac OS For QuickTime Programmers* for more information about the Component Manager and how to use components.

Connecting to the Base Image Decompressor

To use the services of the base image decompressor, your image decompressor component must open a connection to the base image decompressor component. Listing 21-1 illustrates how to make the connection.

Listing 21-1 Connecting to the base image decompressor component

```
CodecComponent baseCodec;
OSErr err;

err = OpenADefaultComponent (decompressorComponentType,
                             kBaseCodecType,
                             &baseCodec);

err = ComponentSetTarget (baseCodec,
                          self);
```

Providing Storage for Frame Decompression

Your image decompressor component uses an `ImageSubCodecDecompressRecord` structure to store information needed to decompress a single frame. The structure is created by the base decompressor component when your component is initialized, as described in "Initializing Your Decompressor Component" (page 591).

Initializing Your Decompressor Component

The first function call that your image decompressor component receives from the base image decompressor is always a call to `ImageCodecInitialize`. In response to this call, your image decompressor component returns an `ImageSubCodecDecompressCapabilities` structure that specifies its capabilities. This structure contains the following fields:

- `canAsync` specifies whether the component can support asynchronous decompression operations, as described in "Asynchronous Decompression" (page 223).
- `decompressRecordSize` specifies the size of the `ImageSubCodecDecompressRecord` structure that the base decompressor component creates for your image decompressor component.

With the help of the base image decompressor, any image decompressor that uses only interrupt-safe calls for decompression operations can support asynchronous decompression.

Listing 21-2 shows how to specify that a decompressor supports asynchronous decompression operations.

Listing 21-2 Specifying the capabilities of a decompressor component

```
ImageSubCodecDecompressCapabilities deccap;
ImageSubCodecDecompressRecord decrec;

deccap->decompressRecordSize = sizeof(decrec);
deccap->canAsync = true;
```

Specifying Other Capabilities of Your Component

The base image decompressor gets additional information about the capabilities of your image decompressor component by calling your component's `ImageCodecPreflight` function. The base image decompressor uses this information when responding to a call to the `ImageCodecPredecompress` function, which the Image Compression Manager makes before decompressing an image.

Your image decompressor component returns information about its capabilities by filling in the `capabilities` structure. Listing 21-3 illustrates how to fill in this structure. In this example, the decompressor component specifies that it supports the ARGB, ABGR, BGRA, and RGBA pixel formats used by Microsoft Windows.

Listing 21-3 Sample implementation of `ImageCodecPreflight`

```
pascal ComponentResult ImageCodecPreflight (
                         ComponentInstance ci,
                         CodecDecompressParams *p)
{
    register CodecCapabilities*capabilities = p->capabilities;

    /*  Decide which depth compressed data we can deal with. */

    switch ( (*p->imageDescription)->depth )  {
        case 16:
            break;
        default:
            return(codecConditionErr);
            break;
    }

    /*  We can deal only 32 bit pixels. */
    capabilities->wantedPixelSize = 32;

    /*  The smallest possible band we can do is 2 scan lines. */

    capabilities->bandMin = 2;

    /*  We can deal with 2 scan line high bands. */

    capabilities->bandInc = 2;

    /*  If we needed our pixels to be aligned on some integer
     *  multiple we would set these to
     *  the number of pixels we need the dest extended by.
     *  If we dont care, or we take care of
     *  it ourselves we set them to zero.
     */

    capabilities->extendWidth = p->srcRect.right & 1;
    capabilities->extendHeight = p->srcRect.bottom & 1;

    {
        OSType *pf = *glob->wantedDestinationPixelTypeH;

        p->wantedDestinationPixelTypes =
            glob->wantedDestinationPixelTypeH;

        // set up default order
        pf[0] = k32BGRAPixelFormat;
        pf[1] = k32ARGBPixelFormat;
        pf[2] = k32ABGRPixelFormat;
        pf[3] = k32RGBAPixelFormat;

        switch (p->dstPixMap.pixelFormat) {
        case k32BGRAPixelFormat: // we know how to do these pixel formats
            break;
        case k32ABGRPixelFormat:
            pf[0] = k32ABGRPixelFormat;
            pf[1] = k32BGRAPixelFormat;
            pf[2] = k32ARGBPixelFormat;
            pf[3] = k32RGBAPixelFormat;
            break;
        case k32ARGBPixelFormat:
            pf[0] = k32ARGBPixelFormat;
            pf[1] = k32BGRAPixelFormat;
            pf[2] = k32ABGRPixelFormat;
            pf[3] = k32RGBAPixelFormat;
            break;
        case k32RGBAPixelFormat:
            pf[0] = k32RGBAPixelFormat;
            pf[1] = k32BGRAPixelFormat;
            pf[2] = k32ARGBPixelFormat;
            pf[3] = k32ABGRPixelFormat;
            break;
        default:            // we don't know how to do these, so return
                            // the default
            break;
        }
    }

    return(noErr);
}
```

Implementing Functions for Queues

If the image decompressor component supports asynchronous scheduled decompression, it receives a `ImageCodecQueueStarting` call from the base image decompressor when processing of the queue begins and the `ImageCodecQueueStopping` function when processing of the queue is finished. It is not necessary for your image decompressor component to implement these functions. Implement them only if there are tasks that your image decompressor component must perform after being notified, such as locking structures in memory before `ImageCodecDrawBand` is called.

`ImageCodecQueueStarting` and `ImageCodecQueueStopping` calls are never made during interrupt time.

Decompressing Bands

Your image decompressor component must implement the `ImageCodecBeginBand` and `ImageCodecDrawBand` functions for decompressing bands. It can also implement the `ImageCodecEndBand` function to be information that decompression of a band is complete. It receives these calls from the base image decompressor when decompression of either a complete frame or an individual band needs to be performed.

Implementing ImageCodecBeginBand

The `ImageCodecBeginBand` function allows your image decompressor component to save information about a band before decompressing it. For example, your image decompressor component can change the value of the `codecData` pointer if not all of the data for the band needs to be decompressed. The base image decompressor preserves any changes your image decompressor component makes to any of the fields in the `ImageSubCodecDecompressRecord` or `CodecDecompressParams` structures.

The `ImageCodecBeginBand` function is never called at interrupt time. If your component supports asynchronous scheduled decompression, it may receive more than one `ImageCodecBeginBand` call before receiving an `ImageCodecDrawBand` call.

A sample implementation of `ImageCodecBeginBand` is shown in Listing 21-4.

Listing 21-4 Sample implementation of `ImageCodecBeginBand`

```
pascal ComponentResult ImageCodecBeginBand (
                          ComponentInstance ci,
                          CodecDecompressParams *p,
                          ImageSubCodecDecompressRecord *drp,
                          long flags)
{
    ExampleDecompressRecord *mydrp = drp->userDecompressRecord;
    long            numLines,numStrips;
    long            stripBytes;
    short           width;
    short           y;
    OSErr           result = noErr;
    Ptr             cDataPtr;

    /* initialize some local variables */

    width = (*p->imageDescription)->width;
    numLines = p->stopLine - p->startLine;  /* number of scanlines in */
                                            /* this band */
    numStrips = (numLines+1)>>1;    /* number of strips in this band */
    stripBytes = ((width+1)>>1) * 5;    /* number of bytes in one */
                                        /* strip of blocks */
    cDataPtr = drp->codecData;

    /*
     *  If skipping some data, just skip it here. We can tell because
     *  firstBandInFrame says this is the first band for a new frame, and
     *  if startLine is not zero, then that many lines were clipped out.
     */
    if ( (p->conditionFlags & codecConditionFirstBand) && p->startLine != 
0 ) {
        if ( p->dataProcRecord.dataProc ) {
            for ( y=0; y  < p->startLine>>1; y++ )  {
                if ( 
(result=CallICMDataProc(p->dataProcRecord.dataProc,&cDataPtr,stripBytes,
                        drp->dataProcRecord.dataRefCon)) != noErr ) {
                    result = codecSpoolErr;
                    goto bail;
                }
                cDataPtr += stripBytes;
            }
        } else
            cDataPtr += (p->startLine>>1) * stripBytes;
    }

    drp->codecData = cDataPtr;
    mydrp->width = width;
    mydrp->numStrips = numStrips;
    mydrp->srcDataIncrement = stripBytes;
    mydrp->baseAddrIncrement = drp->rowBytes<<1;
    mydrp->glob = (void *)storage;

    /* figure out our dest pixel format and select the
       correct DecompressStripProc */
    switch(p->dstPixMap.pixelFormat) {
        case 0:     // old case where planebytes
                    // is not set by codecmanager
        case k32ARGBPixelFormat:
            mydrp->decompressStripProc = DecompressStrip32ARGB;
            break;
        case k32ABGRPixelFormat:
            mydrp->decompressStripProc = DecompressStrip32ABGR;
            break;
        case k32BGRAPixelFormat:
            mydrp->decompressStripProc = DecompressStrip32BGRA;
            break;
        case k32RGBAPixelFormat:
            mydrp->decompressStripProc = DecompressStrip32RGBA;
            break;
        default:
            bail;
            break;
    }

bail:
    return result;
}
```

Implementing ImageCodecDrawBand

When the base image decompressor calls your image decompressor component's `ImageCodecDrawBand` function, your component must perform the decompression specified by the fields of the `ImageSubCodecDecompressRecord` structure. The structure includes any changes your component made to it when performing the `ImageCodecBeginBand` function.

If the `ImageSubCodecDecompressRecord` structure specifies either a progress function or a data-loading function, the base image decompressor never calls the `ImageCodecDrawBand` function at interrupt time. If not, the base image decompressor may call the `ImageCodecDrawBand` function at interrupt time.

If the `ImageSubCodecDecompressRecord` structure specifies a progress function, the base image decompressor handles `codecProgressOpen` and `codecProgressClose` calls, and your image decompressor component must not implement these functions.

If your component supports asynchronous scheduled decompression, it may receive more than one `ImageCodecBeginBand` call before receiving an `ImageCodecDrawBand` call.

A sample implementation of `ImageCodecDrawBand` is shown in Listing 21-5.

Listing 21-5 Sample implementation of `ImageCodecDrawBand`

```
pascal ComponentResult ImageCodecDrawBand (
                          ComponentInstance ci,
                          ImageSubCodecDecompressRecord *drp)
{
    ExampleDecompressRecord *mydrp = drp->userDecompressRecord;
    short y;
    Ptr cDataPtr = drp->codecData;  // compressed data pointer;
    Ptr baseAddr = drp->baseAddr;   // base address of dest PixMap;
    SInt8 mmuMode = true32b;        // we want to be in 32-bit mode
    OSErr err = noErr;

    for (y = 0; y < mydrp->numStrips; y++) {
        if (drp->dataProcRecord.dataProc) {
            if ( (err =
CallICMDataProc(drp->dataProcRecord.dataProc,&cDataPtr,
                        mydrp->srcDataIncrement,
                drp->dataProcRecord.dataRefCon)) != noErr ) {
                err = codecSpoolErr;
                goto bail;
            }
        }

        SwapMMUMode(&mmuMode);      // put us in 32-bit mode

(mydrp->decompressStripProc)(cDataPtr,baseAddr,(short)drp->rowBytes,
                            (short)mydrp->width,glob->sharedGlob);
        SwapMMUMode(&mmuMode);      // put us back

        baseAddr += mydrp->baseAddrIncrement;
        cDataPtr += mydrp->srcDataIncrement;

        if (drp->progressProcRecord.progressProc) {
            if ( (err =
CallICMProgressProc(drp->progressProcRecord.progressProc,
                codecProgressUpdatePercent,
                FixDiv ( y, mydrp->numStrips),
                drp->progressProcRecord.progressRefCon)) != noErr ) {
                err = codecAbortErr;
                goto bail;
            }
        }
    }

bail:
    return err;
}
```

Implementing ImageCodecEndBand

Your image decompressor component is not required to implement the `ImageCodecEndBand` function. If it does, the base image decompressor calls the function when the decompression of a band is complete or is terminated by the Image Compression Manager. The call simply notifies your component that decompression is finished. After your component handles the call, it can perform any tasks that are necessary when decompression is finished, such as disposing of data structures that are no longer used, after receiving notification. Note that because the `ImageCodecEndBand` function can be called at interrupt time, your image decompressor component cannot use this function to dispose of data structures; this must occur after handling the function.

Providing Information About the Decompressor

Your image decompressor component must also implement the `ImageCodecGetCodecInfo`. This performs the same task as the `CDGetCodecInfo` function described in Chapter 4, "Image Compressor Components," in *Inside Macintosh: QuickTime Components*. The Image Compression Manager calls your image decompressor component's `ImageCodecGetCodecInfo` function when it receives a `GetCodecInfo` call.

Providing Progress Information

If the `ImageSubCodecDecompressRecord` structure does not specify a progress function, your image decompressor component can implement `codecProgressOpen` and `codecProgressClose` functions to provide progress information. If the `ImageSubCodecDecompressRecord` structure does specify a progress function, your image decompressor component can implement the `codecProgressUpdatePercent` function to provide progress information during lengthy decompression operations. Implementing this function is optional.

Handling and Delegating Other Calls

Your image decompressor component must delegate the following image decompressor component calls to the base image decompressor:

- `ImageCodecPreDecompress`
- `ImageCodecBandDecompress`
- `ImageCodecBusy`

- ImageCodecFlush

If the ImageSubCodecDecompressRecord structure specifies a progress function, your image decompressor component must also delegate these decompressor component calls to the base image decompressor:

- codecProgressOpen
- codecProgressClose

Your image decompressor component can implement any other image decompressor component functions itself or delegate any of the calls to the base image decompressor. To delegate calls, it uses the DelegateComponentCall function.

Closing the Component

When your image decompressor component closes, it must close its connection to the base image decompressor by calling the CloseComponent function.

Base Image Decompressor Reference

This section describes the data structures and functions for the base image decompressor.

Data Types

The Decompression Capabilities Structure

Your image decompressor component uses the decompression capabilities structure to report its capabilities to the base image decompressor.

```
struct ImageSubCodecDecompressCapabilities
{
    long recordSize; /* size of this record */
    long decompressRecordSize; /* size of decompress record */
    Boolean canAsync; /* true if supports asynch decompression */
};
typedef struct ImageSubCodecDecompressCapabilities
ImageSubCodecDecompressCapabilities;
```

`recordSize`	The size of this `ImageSubCodecDecompressCapabilities` structure, in bytes.
`decompressRecordSize`	The size of the `ImageSubCodecDecompressRecord` structure that your image decompressor component requires. This structure is used to pass information from the `ImageCodecBeginBand` function to the `ImageCodecDrawBand` and `ImageCodecEndBand` functions.
`canAsync`	Specifies whether your image decompressor component can perform asynchronous scheduled decompression. This should be true unless your image decompressor component calls functions that cannot be called during interrupt time.

The Decompression Record Structure

The decompression record structure contains information needed for decompressing a frame.

```
struct ImageSubCodecDecompressRecord
{
    Ptr baseAddr; /* base address of destination pixel map */
    long rowBytes; /* bytes in each row */
    Ptr codecData; /* pointer to compressed data */
    ICMProgressProcRecord progressProcRecord; /* progress function
                                                 structure */
    ICMDataProcRecord dataProcRecord; /* data-loading function
                                         structure */
    void *userDecompressRecord; /* pointer to storage for result */
};
typedef struct ImageSubCodecDecompressRecord
ImageSubCodecDecompressRecord;
```

Field descriptions

`baseAddr` The address of the destination pixel map, which includes adjustment for the offset. Note that if the bit depth of the pixel map is less than 8, your image decompressor component must adjust for the bit offset.

`rowBytes` The offset in bytes from one row of the destination pixel map to the next. The value of the `rowBytes` field must be less than $4000.

`codecData` A pointer to the data to be decompressed.

`progressProcRecord`

A structure that specifies a progress function. This function reports on the progress of a decompression operation. For more information about progress functions, see the "Image Compression Manager"chapter in *Inside Macintosh: QuickTime*.

Here is the definition of the structure:

```
struct ProgressProcRecord
{
    ProgressProcPtr progressProc; /* ptr to progress
                                     function */
long progressRefCon; /* reference constant
                        used by function */
};
typedef struct ProgressProcRecord ProgressProcRecord;
```

If there is no progress function, the Image Compression Manager sets the `progressProc` field in this structure to `nil`.

`dataProcRecord` A structure that specifies a data-loading function. If the data to be decompressed is not all in memory, your component can call this function to load more data. For more information about data-loading functions, see the "Image Compression Manager"chapter in *Inside Macintosh: QuickTime*.

Here is the definition of the structure:

```
struct DataProcRecord
{
    DataProcPtr dataProc; /* pointer to data-loading
                             function */
    long dataRefCon;      /* reference constant
                             used by function */
};
typedef struct DataProcRecord DataProcRecord;
```

If there is no data-loading function, the Image Compression Manager sets the `dataProc` field in the `DataProcRecord` structure to `nil`, and the entire image must be in memory at the location specified by the `codecData` field of the `ImageSubCodecDecompressRecord` structure.

`userDecompressRecord`

A pointer to storage for the decompression operation. The storage is allocated by the base image decompressor after it calls the `ImageCodecInitialize` function. The size of the storage is determined by the `decompressRecordSize` field of the `ImageSubCodecDecompressCapabilities` structure that is returned by the `ImageCodecInitialize` function.

Your image decompressor component should use this storage to store any additional information needed about the frame in order to decompress it.

Functions

This section describes the functions an image decompressor component must include to use the base image decompressor.

ImageCodecBeginBand

The base image decompressor calls your image decompressor component's `ImageCodecBeginBand` function before drawing a band or frame. It allows your image decompressor component to save information about a band before decompressing it. Your component must implement this function.

```
pascal ComponentResult ImageCodecBeginBand (
                    ComponentInstance ci,
                    CodecDecompressParams *params,
                    ImageSubCodecDecompressRecord *drp,
                    long flags);
```

`ci`	Specifies the instance of the image decompressor component for the request.
`params`	Contains a pointer to a decompression parameters structure.
`drp`	Contains a pointer to a decompression record structure.
`flags`	Currently unused; always set to 0.

DISCUSSION

The base image decompressor never calls the `ImageCodecBeginBand` function at interrupt time.

You image decompressor component receives the address of the destination pixel map in the `baseAddr` field of the `drp` parameter. This address includes adjustment for the offset. Note that if the bit depth of the pixel map is less than 8, your image decompressor component must adjust for the bit offset.

The `codecData` field of the `drp` parameter contains a pointer to the compressed video data.

The `userDecompressRecord` field of the `drp` parameter contains a pointer to storage for the decompression operation. The storage is allocated by the base image decompressor after it calls the `ImageCodecInitialize` function. The size of the storage is determined by the `decompressRecordSize` field of the `ImageSubCodecDecompressCapabilities` structure that is returned by the `ImageCodecInitialize` function. Your image decompressor component should use this storage to store any additional information needed about the frame in order to decompress it.

Changes your image decompressor component makes to the `ImageSubCodecDecompressRecord` or `CodecDecompressParams` structures are preserved by the base image decompressor. For example, if your component does not need to decompress all of the data, it can change the pointer to the data to be decompressed that is stored in the `codecData` field of the `ImageSubCodecDecompressRecord` structure.

If your component supports asynchronous scheduled decompression, it may receive more than one `ImageCodecBeginBand` call before receiving an `ImageCodecDrawBand` call.

ImageCodecDrawBand

The base image decompressor calls your image decompressor component's `ImageCodecDrawBand` function to decompress a band or frame. Your component must implement this function.

```
pascal ComponentResult ImageCodecDrawBand (
                        ComponentInstance ci,
                        ImageSubCodecDecompressRecord *drp);
```

`ci`	Specifies the instance of the image decompressor component for the request.
`drp`	Contains a pointer to a decompression record structure.

DISCUSSION

If the `ImageSubCodecDecompressRecord` structure specifies a progress function or data-loading function, the base image decompressor never calls the `ImageCodecDrawBand` function at interrupt time. If not, the base image decompressor may call the `ImageCodecDrawBand` function at interrupt time.

When the base image decompressor calls your image decompressor component's `ImageCodecDrawBand` function, your component must perform the decompression specified by the fields of the `ImageSubCodecDecompressRecord` structure. The structure includes any changes your component made to it when performing the `ImageCodecBeginBand` function.

If your component supports asynchronous scheduled decompression, it may receive more than one `ImageCodecBeginBand` call before receiving an `ImageCodecDrawBand` call.

ImageCodecEndBand

The `ImageCodecEndBand` function notifies your image decompressor component that decompression of a band has finished or that it was terminated by the

Image Compression Manager. Your image decompressor component is not required to implement the `ImageCodecEndBand` function.

```
pascal ComponentResult ImageCodecEndBand (
                    ComponentInstance ci,
                    ImageSubCodecDecompressRecord *drp,
                    OSErr result,
                    long flags);
```

`ci`	Specifies the instance of the image decompressor component for the request.
`drp`	Contains a pointer to a decompression record structure.
`result`	Contains a result code.
`flags`	Currently unused; must be set to 0.

DISCUSSION

The base image decompressor may call the `ImageCodecEndBand` function at interrupt time.

After your image decompressor component handles an `ImageCodecEndBand` call, it can perform any tasks that are required when decompression is finished, such as disposing of data structures that are no longer needed. Because this function can be called at interrupt time, your component cannot use this function to dispose of data structures; this must occur after handling the function.

The value of the `result` parameter is `noErr` if the band or frame was drawn successfully. If it is any other value, the band or frame was not drawn.

RESULT CODES

`noErr`	0	The band or frame was drawn successfully.
	–1	Drawing of the frame was cancelled in response to a call to `CDSequenceFlush`.
`codecAbortErr`	–8967	The operation was aborted by the progress function.

ImageCodecInitialize

The base image decompressor calls your image decompressor component's `ImageCodecInitialize` function before making any other all calls to your component.

Your component must implement this function. It responds by returning an `ImageSubCodecDecompressCapabilities` structure that specifies its capabilities.

```
pascal ComponentResult ImageCodecInitialize (
                    ComponentInstance ci,
                    ImageSubCodecDecompressCapabilities *cap);
```

`ci`	Specifies the instance of the image decompressor component for the request.
`cap`	Specifies the capabilities of the image decompressor component.

DISCUSSION

The`ImageSubCodecDecompressCapabilities` structure that your component returns contains the following fields:

- `canAsync` specifies whether your component can support asynchronous decompression operations, as described in "Asynchronous Decompression" (page 223).
- `decompressRecordSize` specifies the size of the decompression record structure for your component.

ImageCodecPreflight

The base image decompressor calls your image decompressor component's `ImageCodecPreflight` function before decompressing an image. It makes the call when responding to an `ImageCodecPredecompress` call from the Image Compression Manager.

Your component must implement this function. It responds by returning information about its capabilities in a compressor capabilities structure.

```
pascal ComponentResult ImageCodecPreflight (
                        ComponentInstance ci,
                        CodecDecompressParams *params);
```

`ci`	Specifies the instance of the image decompressor component for the request.
`params`	Contains a pointer to a decompression parameters structure.

DISCUSSION

The Image Compression Manager creates the decompression parameters structure, and your image decompressor component is required only to provide values for the `wantedDestinationPixelSize` and `wantedDestinationPixelTypes` fields of the structure. Your image decompressor component can also modify other fields if necessary. For example, if it can scale images, it must set the `codecCapabilityCanScale` flag in the `capabilities` field of the structure. See "The Compressor Capability Structure" (page 234) and "The Compressor Capability Structure" in Chapter 4 of *Inside Macintosh: QuickTime Components* for information about this structure.

ImageCodecQueueStarting

If your component supports asynchronous scheduled decompression, the base image decompressor calls your image decompressor component's `ImageCodecQueueStarting` function before decompressing the frames in the queue.

Your component is not required to implement this function. It can implement the function if it needs to perform any tasks at this time, such as locking data structures.

```
pascal ComponentResult ImageCodecQueueStarting (
                        ComponentInstance ci);
```

`ci`	Specifies the instance of the image decompressor component for the request.

DISCUSSION

The base image decompressor never calls the `ImageCodecQueueStarting` function at interrupt time.

ImageCodecQueueStopping

If your image decompressor component supports asynchronous scheduled decompression, the `ImageCodecQueueStopping` function notifies your component that the frames in the queue have been decompressed. Your component is not required to implement this function.

```
pascal ComponentResult ImageCodecQueueStopping (
                        ComponentInstance ci);
```

`ci`	Specifies the instance of the image decompressor component for the request.

DISCUSSION

The base image decompressor may call the `ImageCodecQueueStopping` function at interrupt time.

After your image decompressor component handles an `ImageCodecQueueStopping` call, it can perform any tasks that are required when decompression of the frames is finished, such as disposing of data structures that are no longer needed. Because this function can be called at interrupt time, your component cannot use this function to dispose of data structures; this must occur after handling the function.

QuickTime Video Effects

This chapter introduces you to QuickTime video effects, which are new in QuickTime 3. You can use “real-time” video effects to control the visual transition between two sources. Sources can be frames of a QuickTime movie, or they can be graphics worlds containing arbitrary images. You can also use effects to visually “filter” a single source, applying a visual effect to a single image.

Because visual effects are calculated and executed at runtime, they can be applied between any two time points, even if the exact appearance of a QuickTime movie at a certain time is not known in advance. This means, for example, you can execute effects on sprite tracks, which can change as a result of user interactions.

You need to read this chapter if you are writing an application that creates QuickTime movies and you want to add video effects to those movies. This is explained in the section “Adding Video Effects to a QuickTime Movie” (page 611).

You should read this chapter if you want to use video effects on graphics worlds, without creating a QuickTime movie, as described in the section “Using Video Effects Outside a QuickTime Movie” (page 631).

If you want to create new video effects of your own, you need to read the section “Creating New Video Effects” (page 638).

In creating QuickTime movies with video effects, you will want to provide the user with ways to choose which effects to apply, when to apply them and how to customize the effects chosen. These topics are discussed in the section “Video Effects User Interface” (page 622).

This chapter also contains two valuable reference sections:

- “Built-in QuickTime Video Effects” (page 682) describes the set of video effects built into QuickTime 3, including implementations of the 133 standard effects defined by the Society of Motion Picture & Television Engineers

(SMPTE), as well as set of useful extra effects implemented by Apple Computer, Inc.

- "QuickTime Video Effects Reference" (page 716) describes the constants, data types, and functions related to QuickTime video effects.

Introduction to QuickTime Video Effects

QuickTime video effects are implemented as Component Manager components, which is the standard mechanism used to extend QuickTime. Effect components are actually a specialized type of image decompressor component. For more information about implementing image decompressor components, see Chapter 4, "Image Compressor Components."

To use an effect component in a QuickTime movie, you add an **effect track** to the movie. The size and duration of the track determines the area of the movie that is effected, and how long the effect runs for.

The effect track has two important attributes: the effect description and the input map. The **effect description** is a sample added to the media of the effects track that selects which of the effect components will be used and contains the arguments to that effect. The **input map** describes the sources that the effect will work on—in other words, which of the other tracks in the movie are effected.

Once you have added the effects track to your movie, QuickTime automatically executes the effect for you when you run the movie.

You can also use the QuickTime video effects outside the context of a QuickTime movie. You still supply an effect description, but instead of creating an effect track, you must write code that executes the individual steps of the effect. For details of this, see the section "Using Video Effects Outside a QuickTime Movie" (page 631).

Adding Video Effects to a QuickTime Movie

This section explains the steps you need to take in order to add video effects to a QuickTime movie. The process is straightforward enough. In brief, you proceed as follows:

1. You add a new **effect track** to your movie: this will contain a sample that defines the characteristics of the effect you want to add.
2. You create an **effect description** that names the particular effect you want, and supplies values for any parameters the effect has.
3. You create an **input map** that defines which tracks in the movie are effected by the effect.
4. Finally, you add the effect description as a new sample to the media of the effect track—this defines how long the effect will run for. As part of this process, you create a **sample description**, which describes the sample being added.

Creating an Effects Track

The first step in adding effects to a QuickTime movie is to create an effects track. This is accomplished by using the standard QuickTime API for track creation, for example:

```
theEffectsTrack = NewMovieTrack(theMovie, kTrackWidth, kTrackHeight, 0);
```

You then call the `NewTrackMedia` function to add a media to the track. The type of the media for an effects track should always be `VideoMediaType`, and the media should have whatever duration you want the effect to have. Here is an example call to `NewTrackMedia`:

```
theEffectsMedia = NewTrackMedia(theEffectsTrack, VideoMediaType, 600,
nil, 0);
```

Creating an Effect Description

The QuickTime 3 SDK includes over 145 transitions, and with QuickTime's extensible architecture, you can easily add many additional effects. Each effect has a set of parameters that controls how the effect appears at runtime. You need a way to choose an effect and specify appropriate values for its parameters.

To do this, you create an effect description that selects the effect from the available range, and set its parameters to produce the exact effect you require.

Structure of an Effect Description

An effect description is a `QTAtomContainer` data structure, the general QuickTime structure for holding a set of `QTAtoms`. Each `QTAtom` in the effect description contains the value for a single parameter of the effect. All effect descriptions must contain the set of **required atoms**, which specify attributes such as which effect component will be used. In addition, effect descriptions can contain a variable number of **parameter atoms**, which hold the values of the parameters of the effect.

Each atom contains either data or a set of child atoms. If an atom contains data, the data is the value of the attribute or parameter, and this value remains constant while the effect executes. If the atom contains a set of child atoms, they should contain a **tween entry** so the value of the parameter will be interpolated for the duration of the effect.

For detailed information about creating and editing `QTAtom` and `QTAtomContainer` data structures, see Chapter 1, "Movie Toolbox." For more information about tweens, see Chapter 25, "Tween Components and Native Tween Types."

You assemble an effect description by adding the appropriate set of atoms to a `QTAtomContainer` data structure. You then add the entire description to the media of the video track that contains the effect. The effect description forms the sample for the media of the video track.

Required Atoms of an Effects Description

There are several required atoms that an effect description must contain. The first is the `kParameterWhatName` atom that contains the name of the effect. This selects which of the available effects will be used.

The code snippet shown in Listing 22-1 adds a `kParameterWhatName` atom to the atom container `effectDesc`. The constant `kCrossFadeTransitionType` contains the name of the built-in cross fade effect. The cross fade effect is described in detail in "Cross Fade ('dslv')" (page 701).

Listing 22-1 Adding a `kParameterWhatName` atom with the value `'kCrossFadeTransitionType` to the `QTAtomContainer effectDesc`

```
effectCode = kCrossFadeTransitionType;
QTInsertChild(effectDescription,
                kParentAtomIsContainer,
                kParameterWhatName,
                kParameterWhatID,
                0,
                sizeof(effectCode),
                &effectCode,
                nil);
```

In addition to the `kParameterWhatName` atom, the effect description must also contain zero or more `kEffectSourceName` atoms. Each of these atoms contains the name of one of the effect's sources. The input map is a data structure that maps these names to the actual tracks of the movie that are the sources. "Creating an Input Map" (page 615) describes how to create the input map.

Parameter Atoms of an Effects Description

In addition to the required atoms, the effects description contains a variable number of parameter atoms. The number and types of parameter atoms vary from effect to effect. For example, the built-in cross fade effect has only one parameter, while the general convolution filter effect has nine. Some effects have no parameters at all, and do not require any parameter atoms. The section "Built-in QuickTime Video Effects" (page 682) describes the parameters expected by the built-in effects.

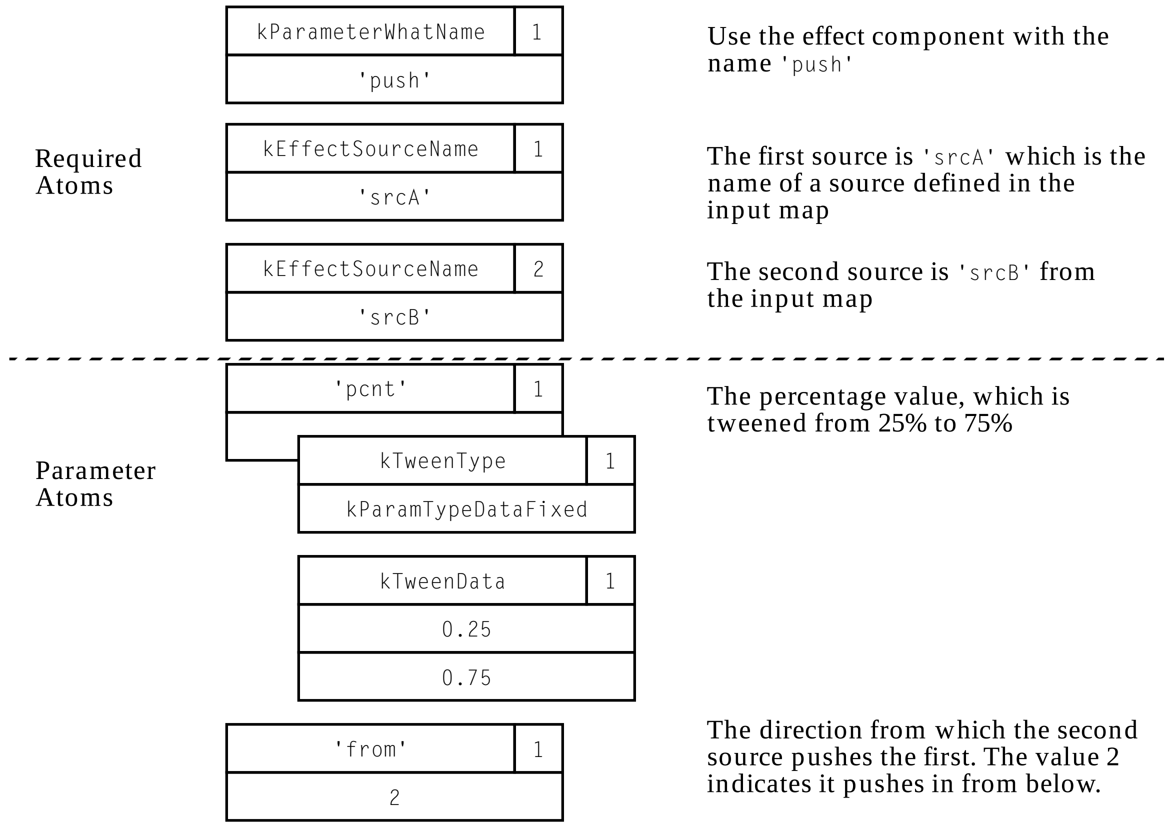
The `QTInsertChild` function is used to add these parameters to the effect description, as seen in the code example in Listing 22-1.

Consider, for instance, the push effect (page 715). In this example, the source `'srcB'` will push in from the bottom, covering the source `'srcA'`. The `'pcnt'` parameter atom defines which frame of the effect is shown. This should always contain a tween entry, so that the value of this parameter is interpolated as the effect runs. This will cause consecutive frames of the effect to be executed, allowing the push effect to take place.

In this example, the effect will start 25% of the way through (with `'srcB'` already partly on screen) and finish 75% of the way through (with part of `'srcA'` still visible).

Figure 22-1 shows the set of atoms that must be added to the entry description.

Figure 22-1 An example effect description for the Push effect


kParameterWhatName
1
'push'
Use the effect component with the name 'push'
Required Atoms
kEffectSourceName
1
'srcA'
The first source is 'srcA' which is the name of a source defined in the input map
kEffectSourceName
2
'srcB'
The second source is 'srcB' from the input map
Parameter Atoms
'pcnt'
1
The percentage value, which is tweened from 25% to 75%
kTweenType
1
kParamTypeDataFixed
kTweenData
1
0.25
0.75
'from'
1
2
The direction from which the second source pushes the first. The value 2 indicates it pushes in from below.

Creating an Input Map

The input map is another `QTAtomContainer` that you attach to the effects track. It describes the sources used in the effect, and names each source. This name is the one that is used to refer to the source in the effect description.

Structure of an Input Map

The input map contains a set of atoms that refer to the tracks used in the effect. These are known as the **reference atoms**. Each track is represented by one `QTAtom` of type `kTrackModifierInput`. The ID of the reference atom is set to the **reference index** of the track being used as the source: you generate the reference index by calling `AddTrackReference` on the track to be indexed.

Each reference atom contains two children, which hold the name and type of the source. The name of the track is a unique identifier that you provide, which is used in the effect description to reference the track. Any four-character name is valid, as long as it is unique in the set of source names.

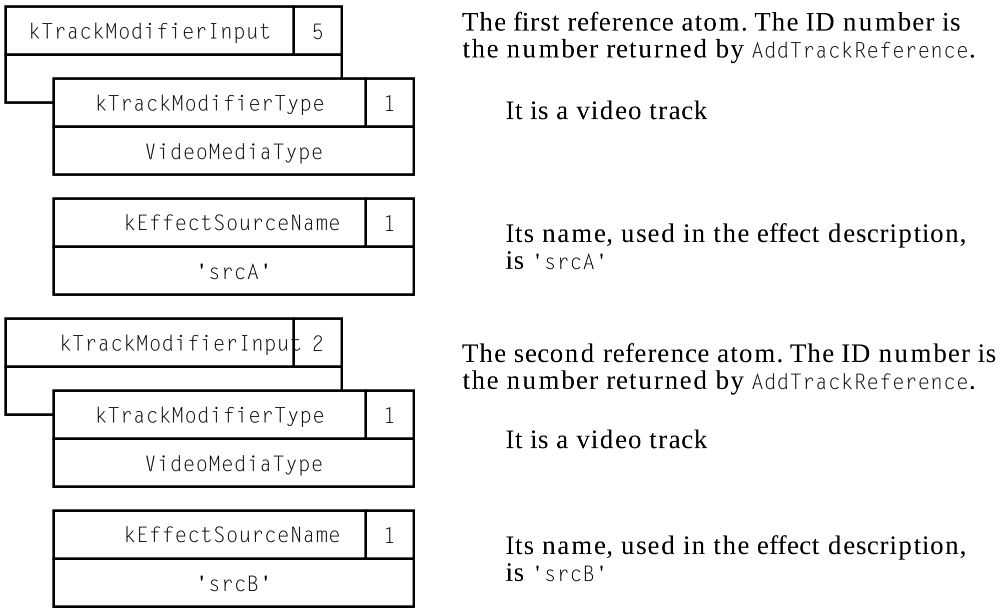
IMPORTANT

Apple recommends you adopt the standard naming convention `'srcX'`, where X is a letter of the alphabet. Thus, your first source would be named `'srcA'`, the second `'srcB'` and so forth. This convention is used in this chapter. ▲

The type of a reference atom indicates the type of the track being referenced. For a video track the type is `VideoMediaType`, for a sprite track it is `SpriteMediaType`, and so forth. Video tracks are likely to be the most common track type used as sources for effects. Only tracks that are media tracks and have a visible component, such as video and sprite tracks, can be used as sources for an effect. This means, for example, that sound tracks cannot be sources to an effect.

Figure 22-2 shows a completed input map that references two sources. The first source is a video track, and is called `'srcA'`. The second source, also a video track, is called `'srcB'`. The ID numbers of the two reference atoms are the reference index numbers returned by the call to `AddTrackReference`.

Figure 22-2 An example input map referencing two sources


kTrackModifierInput
5
kTrackModifierType
1
VideoMediaType
kEffectSourceName
1
'srcA'
kTrackModifierInput
2
kTrackModifierType
1
VideoMediaType
kEffectSourceName
1
'srcB'
The first reference atom. The ID number is the number returned by AddTrackReference.
It is a video track
Its name, used in the effect description, is 'srcA'
The second reference atom. The ID number is the number returned by AddTrackReference.
It is a video track
Its name, used in the effect description, is 'srcB'

Building Input Maps

The first step in creating an input map is to create a new `QTAtomContainer` to hold the map. You use the standard QuickTime container creation function:

```
QTNewAtomContainer(&inputMap);
```

For each source you are creating, you need to call the `AddTrackReference` function, which returns a reference index to the track. You will use this reference index as the ID of the reference atom you add to the input map.

The code in Listing 22-2 gets a reference to the track `firstSourceTrack`, and adds it in an atom attached to the newly created input map.

Listing 22-2 Adding a reference in an atom to an input map

```
AddTrackReference(theEffectsTrack,
                  firstSourceTrack,
                  kTrackModifierReference,
                  &referenceIndex);

QTInsertChild(inputMap,
             kParentAtomIsContainer,
             kTrackModifierInput,
             referenceIndex,
             0,
             0,
             nil,
             &inputAtom);
```

You now need to add the name and type of the source track to the reference atom (which has been stored in `inputAtom`). Again, calling the `QTInsertChild` function does this, as in the following source code:

```
inputType = VideoMediaType;
QTInsertChild(inputMap,
             inputAtom,
             kTrackModifierType,
             1,
             0,
             sizeof(inputType),
             &inputType,
             nil);

aType = 'srcA';
QTInsertChild(inputMap,
             inputAtom,
             kEffectSourceName,
             1,
             0,
             sizeof(aType),
             &aType,
             nil);
```

This process is repeated for each track that will act as a source for the effect.

Adding the Effect Description to the Track

The effect description, as explained in “Creating an Effect Description” (page 612), will form the sample that is added to the media of the effects track. Before you can add the sample, you must create a corresponding **sample description**, that is, a data structure that describes the sample being added.

▲ **WARNING**
It is important to distinguish between the effect description and the sample description. The *effect description* is a data structure that describes the effect to be used. The *sample description* is a data structure that describes a sample of a track’s media. The sample description is needed when you add your effect description to an effect track. ▲

To create a sample description, you create and fill out a data structure of type `ImageDescription`. The code shown in Listing 22-3 provides an example of how to do this.

Listing 22-3 Creating and filling out a sample description

```
sampleDesc = (ImageDescriptionHandle)
NewHandleClear(sizeof(ImageDescription));

(**sampleDesc).idSize              = sizeof(ImageDescription);
(**sampleDesc).cType               = 'fmns';
(**sampleDesc).vendor              = 'appl';
(**sampleDesc).temporalQuality     = codecNormalQuality;
(**sampleDesc).spatialQuality      = codecNormalQuality;
(**sampleDesc).width               = 640;
(**sampleDesc).height              = 480;
(**sampleDesc).hRes                = 72L << 16;
(**sampleDesc).vRes                = 72L << 16;
(**sampleDesc).frameCount          = 1;
(**sampleDesc).depth               = 24;
(**sampleDesc).clutID              = -1;
```

The `ImageDescription` data structure is described in detail in Chapter 4, “Image Compressor Components.”

The `width`, `height`, `vRes`, `hRes`, `depth` and `clutID` fields describes the spatial characteristics of the sample. In this example, the effect sample is 640x480 pixels, has a resolution of 72 dpi (the standard screen resolution), runs at 24-bits and does not have a color lookup table.

The `spatialQuality` and `temporalQuality` fields describe the quality of the effect.

The `cType` and `vendor` fields specify which of the currently installed effects will be used to decompress this sample. The `cType` field holds the name of the effect component to be used. The `vendor` field should contain the code for the supplier of the effect component you want to use. Because no central registry of effect names exists, suppliers could use the same effect code to identify their effect components; using the `vendor` field allows you to make sure that the right component is selected.

In the example in Listing 22-3, the film noise ('`fmns`') effect from Apple ('`appl`') has been chosen.

Note
If the specified effect component is not available when the sample is decompressed (i.e., when the movie containing the effect track is run) an error will occur. ◆

Adding the Sample to the Media

Once you have the sample description prepared, you call `AddMediaSample` to add the effect description to the media. Listing 22-4 shows an example call.

Listing 22-4 Calling `AddMediaSample` to add the effect description

```
// Always call BeginMediaEdits before adding sample to a media
BeginMediaEdits(theEffectsMedia);

// Add the sample to the media
AddMediaSample(theEffectsMedia,
                (Handle) theEffectDescription,
                0,
                GetHandleSize((Handle) theEffectDescription),
                600,
                (SampleDescriptionHandle) sampleDescription,
                1,
                0,
                &sampleTime);

// End the media editing session
EndMediaEdits(theEffectsMedia);
```

Adding the Input Map

Adding the input map to the effects track is easy. The function `SetMediaInputMap` adds an input map to a track's media. Continuing from the code sample in Listing 22-4, you call:

```
SetMediaInputMap(theEffectsMedia, theInputMap);
```

Finishing Up

Finally, you insert the media, complete with sample and input map, into the track, using the `InsertMediaIntoTrack` call, as follows:

```
InsertMediaIntoTrack(theEffectsTrack,
                        0,
                        sampleTime,
                        GetMediaDuration(theEffectsMedia),
                        fixed1);
```

Video Effects User Interface

If your application creates QuickTime movies with video effects, you will want to provide the user with ways to choose which effects to apply, when to apply them and how to customize the effects chosen. Although you are free to write your own code for these user interface tasks, you are encouraged to use the APIs that QuickTime provides to assist with them.

QuickTime provides a standard dialog box (called the **standard parameters dialog box**) that allows the user to select an effect and then choose values for its parameters. Using this dialog box means that your users will see an interface that is standard across applications. In addition, if effects parameters change in the future, your application will not need to be rebuilt to use them. Installing the latest version of QuickTime will do this for you.

In many applications, it is appropriate to simply show the standard parameters dialog box and let users choose and customize effects with it. QuickTime provides a set of high-level API functions that you can call to achieve this. “Displaying the Effects User Interface Using the High-Level API” (page 622) takes you through the use of these high-level functions.

Sometimes you need greater control over the effects user interface. You might, for example, want to add the controls from the standard parameters dialog box to one of your application’s own dialog boxes. QuickTime provides a set of low-level APIs that let you do this. They give you greater control and flexibility, but they are more complex to use than the high-level APIs. These low-level functions are discussed in “Adding Video Effects Controls to an Existing Dialog Box” (page 627).

Displaying the Effects User Interface Using the High-Level API

This section describes the set of functions you can call to use the standard parameters dialog box in your applications.

Getting a List of Effects

Before your application can present the standard parameters dialog box to users, you need to build a list of the effects that are available. QuickTime provides the `QTGetEffectsList` function to do this for you. This returns a `QTAtomContainer` that contains a list of all the effects currently available.

▲ **WARNING**
This function can take several seconds to execute, so you should typically only call it once when your application is launched. After that you will only need to reload the effect list after a pair of suspend and resume effects. ▲

You can remove effects from the returned list, if you want to restrict the set of effects the user can choose from.

Displaying the Standard Parameters Dialog Box

Once you have a list of effects, you are ready to ask QuickTime to display a dialog box that allows the user to choose and customize an effect. This is known as the **standard parameters dialog box**. You use the `QTCreateStandardParameterDialog` function to display the dialog box.

Your application can use this function either to display a stand-alone dialog box, or to add the user interface elements from the standard parameter dialog box to one of the application's own dialog boxes. These options are discussed separately in the following sections.

For full details of the `QTCreateStandardParameterDialog` function, see the reference section "QTCreateStandardParameterDialog" (page 727).

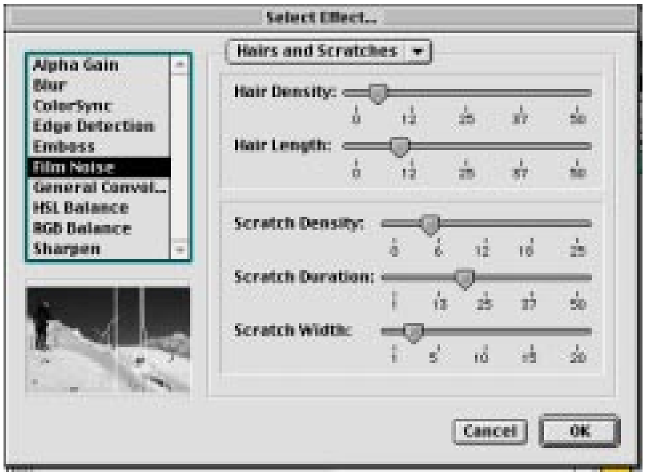
Displaying the Standard Parameters Dialog Box Directly

When you call `QTCreateStandardParameterDialog`, QuickTime creates a standard parameters dialog box. The contents of the dialog box will vary, depending on the list of effects you passed to the function and the set of parameters that the currently chosen effect has.

Calling `QTCreateStandardParameterDialog` does not display the standard parameters dialog box, it only prepares it for display. The dialog box is shown the first time you call `QTIsStandardParameterDialogEvent` to process events for the dialog box. Event handling is described in "Processing Standard Parameter Dialog Box Events" (page 625).

The standard parameters dialog box for Apple's film noise effect is shown in Figure 22-3.

Figure 22-3 The standard parameters dialog box showing Apple's Film Noise effect



Notice that the dialog box has three main sections. In the upper left corner is a scrolling list of all the components. To the right of this are displayed the parameters of the chosen effect. As you select different items in the list of effects, the parameters change appropriately. Finally, there is the effect preview, which is below the list of effects. This shows a preview of the currently chosen effect and parameter settings applied to a short preview movie clip.

The function call to create and display this dialog box is:

```
QTCreateStandardParameterDialog(theEffectList,
                                theEffectParameters,
                                0,
                                &createdDialogID);
```

The variable `theEffectList` holds list of effects that was returned by `QTGetEffectsList`. You can also pass `nil` for this value, in which case `QTCreateStandardParameterDialog` calls `QTGetEffectsList` to generate the list of all the currently installed effects, then shows these effects. The argument `theEffectParameters` contains a `QTAtomContainer` that holds the initial values of the effect's parameters. In most cases, you should pass an empty `QTAtomContainer` as this argument, in which case the default values of each effect are shown. When the user selects the dialog box's OK button, the chosen effect's parameter values are written into `theEffectParameters`.

The third argument contains dialog options. See the reference section "QTCreateStandardParameterDialog" (page 727) for details of the possible options values that can be passed.

The `createdDialog` argument returns an ID number that is passed to the other functions that deal with the standard parameters dialog box. This is explained in detail in the next section.

Processing Standard Parameter Dialog Box Events

Once the dialog box has been created, you must process events sent to it using an event loop. You repeatedly call `WaitNextEvent` and pass the events returned through the `QTIsStandardParameterDialogEvent` function.

The `QTIsStandardParameterDialogEvent` function checks each event to see if it relates to the standard parameters dialog box. You should continue to handle events that are not related to the standard parameters dialog box as normal.

▲ **WARNING**
You should not use the `ModalDialog` function to process events for a standard parameters dialog box. The `ModalDialog` function is not guaranteed to work correctly in all circumstances with a standard parameters dialog box. ▲

You pass the event record returned from `WaitNextEvent` to the function `QTIsStandardParameterDialogEvent`, then check the return value to find out how the events was handled. The possible return values are:

- `noErr` – the event was related to the standard parameters dialog box and was completely processed. Your application should not process this event, instead it should poll `WaitNextEvent` again.
- `featureUnsupported` – the event was not related to the standard parameters dialog box. Your application should process the event in the normal way
- `codecParameterDialogConfirm` – the user clicked the OK button in the standard parameters dialog box. Your application should call `QTDismissStandardParameterDialog` to close the dialog box. The values chosen by the user are put into the variable you passed in the `parameters` parameter when you called `QTCreateStandardParameterDialog`.

- `userCanceledErr` – the user clicked the Cancel button in the standard parameters dialog box. Your application should call `QTDismissStandardParameterDialog` to close the dialog box.

The `QTIsStandardParameterDialogEvent` function may also return error codes, as described in "QuickTime Video Effects Reference" (page 716). Your application should only process the event returned from `WaitNextEvent` if `QTIsStandardParameterDialogEvent` returns `featureUnsupported`.

The code in Listing 22-5 is an example event loop showing how `QTIsStandardParameterDialogEvent` is used.

Listing 22-5 An example event loop showing use of `QTIsStandardParameterDialogEvent`

```
while (result == noErr)
{
    EventRecord     theEvent;

    WaitNextEvent(everyEvent, &theEvent, 0, nil);
    result = QTIsStandardParameterDialogEvent(&theEvent,
                                            createdDialogID);
    switch (result)
    {
        case featureUnsupported:
        {
            result = noErr;

            switch (theEvent.what)
            {
                case updateEvt:
                    BeginUpdate((WindowPtr) theEvent.message);
                    EndUpdate((WindowPtr) theEvent.message);
                    break;
            }
            break;
        }

        case codecParameterDialogConfirm:
        case userCanceledErr:
                QTDismissStandardParameterDialog(createdDialogID);
                createdDialogID = nil;
                break;
        }
    }
}
```

Adding Video Effects Controls to an Existing Dialog Box

In most circumstances, it is easy to use the high-level functions described to present a user interface for the video effects. However, there are occasions when you will need finer control over the way the interface is presented. QuickTime provides a set of low-level API functions for these circumstances.

For example, you may want to incorporate the controls from the standard parameters dialog box into an existing dialog box that your application displays, rather than using a completely independent dialog box. This would be necessary if, for example, you wanted a single dialog box that allowed users to customize an effect and specifying its duration. To do this, you would create a dialog box with a control for the duration of the effect and then, at runtime, you would add the customization controls from the standard parameters dialog box by calling the function `ImageCodecCreateStandardParameterDialog`.

Note

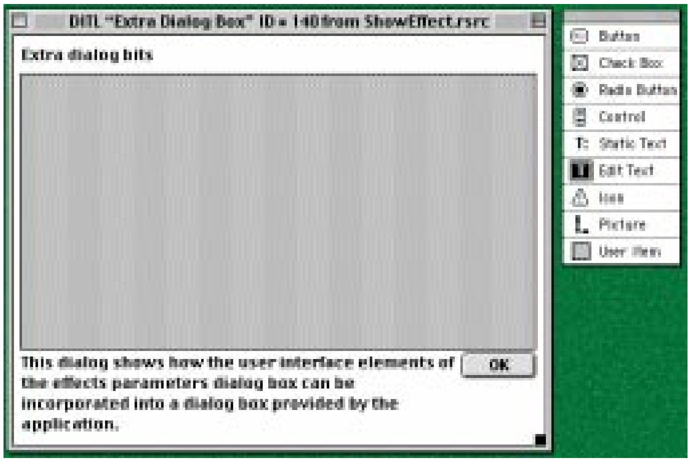
The low-level APIs follow the same naming convention as their high-level counterparts, except that the low-level versions are named `ImageCodec…` where the high-level versions are named `QT…`. ◆

Creating Your Application's Dialog Box

First create the dialog box that will incorporate the controls from the standard parameters dialog box. This dialog box must contain a user item that is large enough to hold the controls that will be added. If you are using Chicago 12 point as your dialog box font, the user item should be 250 pixels wide and 300 pixels high. If you are using Geneva 9 point, the user item should be 150 pixels wide and 200 pixels high.

Figure 22-4 shows a dialog box being edited in ResEdit. The dialog box contains two static text items and an OK button. It also contains a user item that defines the area where the controls from the standard parameters dialog box will appear.

Figure 22-4 A sample application dialog box that will incorporate the effects controls from the standard parameters dialog box.


DITL "Extra Dialog Box" ID = 140 from ShowEffect.rsrc

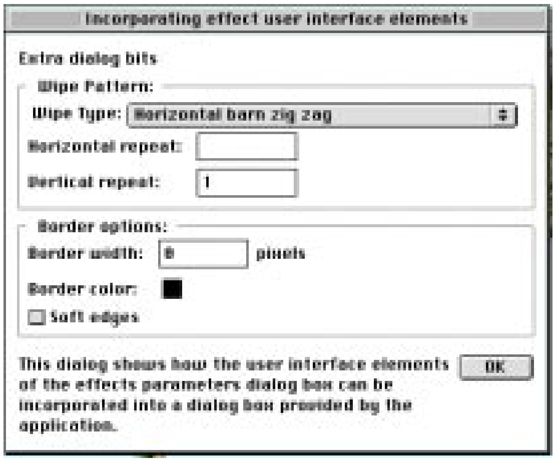
Extra dialog bits

This dialog shows how the user interface elements of the effects parameters dialog box can be incorporated into a dialog box provided by the application.

Figure 22-5 shows the same dialog box in the running application after the effects controls have been incorporated.

Note
You should make sure that the dialog box resource is marked as not initially visible. You will call a QuickTime function to add the effects controls from the standard parameter dialog box to the application's dialog box before showing it. ◆

Figure 22-5 The dialog box shown in Figure 22-4 after incorporation of the effects controls.



Incorporating effect user interface elements

Extra dialog bits

Wipe Pattern:

Wipe Type: Horizontal barn zig zag

Horizontal repeat:

Vertical repeat: 1

Border options:

Border width: 8 pixels

Border color: ■

☐ Soft edges

This dialog shows how the user interface elements of the effects parameters dialog box can be incorporated into a dialog box provided by the application.

Incorporating Controls from the Standard Parameters Dialog Box

Once you have defined the resource that defines your dialog box, you are ready to add code to your application to create the dialog box and add the effects controls to it. First, call `GetNewDialog` in the usual way to create an instance of the dialog box. Then call `ImageCodecCreateStandardParameterDialog` to incorporate the controls from the standard parameter dialog box into the dialog box. You then show the dialog box in the usual way.

The code in Listing 22-6 shows these steps:

Listing 22-6 Creating the dialog box and adding effect controls

```
movableModalDialog = GetNewDialog(kExtraDialogID, nil, (WindowPtr) -1);
if (movableModalDialog!=nil)
{
    // Add the user interface elements from the standard parameters
    // dialog box to the modal dialog box just created
    ImageCodecCreateStandardParameterDialog(gCompInstance,
                            parameterDescription,
                            gEffectSample,
                            pdOptionsModelDialogBox,
                            movableModalDialog,
                            kExtraUserItemID,
                            &createdDialogID);

    // Now show the dialog box and make it the default port
    ShowWindow(movableModalDialog);
    SelectWindow(movableModalDialog);
    SetPort(movableModalDialog);
}
```

The call to `ImageCodecCreateStandardParameterDialog` shows how to pass the existing dialog box (`movableModalDialog`) and the item number of the user item (`kExtraUserItemID`) that will be replaced with the controls from the standard parameter dialog box.

Once the effect controls have been added, the dialog box is shown, selected and made the default port.

The rest of the code needed to handle the dialog box is largely the same as dealing with a stand-alone parameters dialog box. You pass every non-null event returned from `WaitNextEvent` through `ImageCodecIsStandardParameterDialogEvent`, and only continue processing events if it returns `featureUnsupported`.

You track `MouseDown` events sent to the dialog box as normal. When the user clicks the OK button in the dialog box, you need to retrieve the values from the incorporated standard parameters dialog box. To do this, call the function `ImageCodecStandardParameterDialogDoAction` with the `pdActionConfirmDialog` action selector. This retrieves an effect description that describes the parameter values the user has chosen. You then call `ImageCodecDismissStandardParameterDialog` to dispose of the incorporated elements. After this is done, you call `DisposeDialog` to correctly dispose of the application's dialog box.

Adding a Preview to your Dialog Box

You may have noticed that the incorporated controls shown in Figure 22-5 do not include a preview, as the standard parameters dialog box does (Figure 22-3).

In order for your dialog box to show a preview of the effect, include another user item in the application's dialog box, to contain the preview movie clip. Then call the `ImageCodecStandardParameterDialogDoAction` function with the `pdActionSetPreviewUserItem` action selector, as shown in the following code snippet:

```
myErr = ImageCodecStandardParameterDialogDoAction(gCompInstance,
                                        gEffectsDialog,
                                        pdActionSetPreviewUserItem,
                                        (void *) kPreviewUserItemID);
```

This makes the user item whose item number is `kPreviewUserItemID` (an application-defined constant in this example) the previewer for your dialog box.

You can use the `ImageCodecStandardParameterDialogDoAction` function to perform a number of similar customizations; see the reference section "ImageCodecStandardParameterDialogDoAction" (page 737) for more details.

Using Video Effects Outside a QuickTime Movie

"Adding Video Effects to a QuickTime Movie" (page 611) describes creating a QuickTime movie that uses video effects. You can also use QuickTime video effects to transition between two graphics worlds. Your application does not have to generate a QuickTime movie to use the video effects. This section deals with the task of running an effect that transitions between two graphics worlds. The general principles also apply to filtering a single image held in a graphics world.

Preparing to execute an effect outside the context of a QuickTime movie is similar to preparing to add a video effect to a movie: you provide an effect description and a sample description. The main difference is that instead of building an input map to describe the sources used by the effect, you use the function `CDSequenceNewDataSource` to use a graphics world as the source for the effect.

As well as setting up the effect, you must provide code to run it, since QuickTime cannot directly control the playback of the effect. Because effects are just a type of image decompressor, the code to execute an effect is the same code you would use to decompress and display an image sequence.

Preparing an Effect for Direct Execution

The code that prepares the data structures required to directly execute an effect is broadly similar to the code to set up effects within a QuickTime movie.

You first provide an effect description and sample description for the effect component you are going to use. Then you prepare a decompression sequence that will actually playback the effect.

Setting up an Effect Description

The first step is to provide an effect description. The code shown in Listing 22-7 is very similar to the code described in “Creating an Effect Description” (page 612).

Listing 22-7 Providing an effect description

```
{
    QTAtomContainer     myEffectDesc = NULL;
    OSType              myType;
    OSErr               myErr = noErr;
    long                myLong;

    // Create a new, empty effect description
    myErr = QTNewAtomContainer(&myEffectDesc);

    // This example is an effect description for a SMPTE effect
    myType = OSTypeConst('smpt');
    myErr = QTInsertChild(myEffectDesc,
                    kParentAtomIsContainer,
                    kParameterWhatName,
                    kParameterWhatID,
                    0,
                    sizeof(myType),
                    &myType,
                    NULL);

    // Now reference the two sources to be used. These will not be items
    // in an input map. Instead the source names used will be resolved in
    // the decompression sequence.
    myType = OSTypeConst('srcA');
    myErr = QTInsertChild(myEffectDesc,
                    kParentAtomIsContainer,
                    kEffectSourceName,
                    1,
                    0,
                    sizeof(myType),
                    &myType,
                    NULL);

    // Add the second source
    myType = OSTypeConst('srcB');
    myErr = QTInsertChild(myEffectDesc,
                    kParentAtomIsContainer,
                    kEffectSourceName,
                    2,
                    0,
                    sizeof(myType),
                    &myType,
                    NULL);

    // This example uses SMTPE effect number 74, so add a WipeID
    // parameter with the value 74
    myLong = 74;
    myErr = QTInsertChild(myEffectDesc,
                    kParentAtomIsContainer,
                    OSTypeConst('wpID'),
                    1,
                    0,
                    sizeof(myLong),
                    &myLong,
                    NULL);
}
```

Preparing the Sample Description

Next, a sample description is prepared. This is identical to the sample description that would be prepared if the same effect was added to a QuickTime movie. The code in Listing 22-8 creates a sample description that selects Apple's SMPTE wipe (`'smpt'`) effect.

Listing 22-8 Preparing a Sample Description

```
{
    ImageDescriptionHandlemySampleDesc = NULL;

    // create a new, empty sample description
    mySampleDesc =
        (ImageDescriptionHandle)NewHandleClear(sizeof(ImageDescription));

    // fill in the fields of the sample description
    (**mySampleDesc).idSize             = sizeof(ImageDescription);
    (**mySampleDesc).cType              = 'smpt';
    (**mySampleDesc).vendor             = kAppleManufacturer;
    (**mySampleDesc).temporalQuality    = codecNormalQuality;
    (**mySampleDesc).spatialQuality     = codecNormalQuality;
    (**mySampleDesc).width              = theWidth;
    (**mySampleDesc).height             = theHeight;
    (**mySampleDesc).hRes               = 72L << 16;
    (**mySampleDesc).vRes               = 72L << 16;
    (**mySampleDesc).frameCount         = 1;
    (**mySampleDesc).depth              = 0;
    (**mySampleDesc).clutID             = -1;

    return(mySampleDesc);
}
```

Preparing the Decompression Sequence

Because the effect is directly executed, the effect description is not added to a video track, as described in "Adding the Effect Description to the Track" (page 619). Instead, a decompression sequence is created that will execute the effect. The sample, and its description, form part of the input to the decompression sequence. For complete details of setting up and using decompression sequences, see Chapter 4, "Image Compressor Components."

The code shown in Listing 22-9 prepares a decompression sequence for playback, given the effect description in `myEffectDesc` and the sample description in `mySampleDesc`. The function generates a sequence identifier, which in this example is stored in the variable `gEffectSequenceID`. The variable `mainWindow` contains the display surface on which the effect is shown. In this case, it is the application's main window. Full details of this call are provided in Chapter 3, "Image Compression Manager."

Listing 22-9 Preparing a decompression sequence used to execute an effect directly

```
HLock((Handle) myEffectDescription);

DecompressSequenceBeginS(&gEffectSequenceID,
                    mySampleDescription,
                    StripAddress(*myEffectDescription),
                    GetHandleSize(myEffectDescription),
                    (CGrafPtr) mainWindow,
                    NULL,
                    NULL,
                    NULL,
                    ditherCopy,
                    NULL,
                    0,
                    codecNormalQuality,
                    NULL);

HUnlock((Handle) myEffectDescription);
```

The value passed in the `accuracy` field to this call (in this example the value is `codecNormalQuality`) controls how the effect component optimizes the rendering of the effect. Values less than or equal to `codecNormalQuality` indicate that the decompression sequence should be executed in near real-time effect. This means that the visual quality of the effect may be compromised in order to achieve high-speed playback.

A setting of `codecHighQuality` indicates to the component that it should not attempt to sacrifice quality of the rendering for speed of playback, and should instead render the effect with maximum visual quality. This is particularly useful when the effect is being used to process images off-line.

Adding sources to the decompression sequence

Once the decompression sequence is set up, the sources for the effect must be associated with the source names used in the effect description. When you are creating a QuickTime movie containing effects, the input map provides this association. When an effect is being directly executed, use the functions `CDSequenceNewDataSource` and `CDSequenceSetSourceData` to create the named sources.

The code in Listing 22-10 shows how image descriptions for a graphics world (`gWorld1`) are generated by calling `MakeImageDescriptionForPixMap`. The image description is then named `srcA` using the `CDSequenceNewDataSource` function.

Listing 22-10 Generating image descriptions for a graphics world

```
{
    ImageSequenceDataSource     mySrc1 = 0;

    // Generate a description of the first graphics world and store it in
    // gWorld1Desc
    myErr = MakeImageDescriptionForPixMap(gWorld1->portPixMap,
                                          &gWorld1Desc);

    // Create a source from the graphics world description.
    myErr = CDSequenceNewDataSource(gEffectSequenceID,
                            &mySrc1,
                            'srcA',
                            1,
                            (Handle)gWorld1Desc,
                            NULL,
                            0);

    // Set the data for source srcA to be the pixMap of the graphics
    // world gWorld1
    CDSequenceSetSourceData(mySrc1,
                        GetPixBaseAddr(gWorld1->portPixMap),
                        (**gWorld1Desc).dataSize);
}
```

Adding a time base to the decompression sequence

The last step in preparing the decompression sequence is to create a time base that will be used in the playback of the effect.

The code in Listing 22-11 creates a new time base and sets its rate to `0`. The new time base is then associated with the newly created decompression sequence.

The time base's rate is set to 0 because the effect is being played outside the context of a QuickTime movie. This means your application must repeatedly call the RunEffect function (described below) to play the effect, instead of relying on QuickTime to honor the time base rate.

Listing 22-11 Adding a time base to a decompression sequence

```
gTimeBase = NewTimeBase();
SetTimeBaseRate(gTimeBase, 0);
CDSequenceSetTimeBase(gEffectSequenceID, gTimeBase);
```

Executing the Decompression Sequence

With the effect and sample descriptions built and the decompression sequence prepared, you can now execute the effect. The function shown in Listing 22-12 executes a single frame of a decompression sequence.

The parameter theTime contains the number of the frame to be executed. The parameter theNumberOfSteps contains the total number of frames that will be used to run the effect.

Listing 22-12 The RunEffect function, which executes one frame of an effect

```
// Decompress a single step of the effect sequence at time.
OSErr RunEffect(TimeValue theTime, int theNumberOfSteps)
{
    OSErr                   err = noErr;
    ICMFrameTimeRecord      frameTime;

    // Set the timebase time to the step of the sequence to be rendered
    SetTimeBaseValue(gTimeBase, theTime, theNumberOfSteps);

    frameTime.value.lo      = theTime;
    frameTime.value.hi      = 0;
    frameTime.scale         = theNumberOfSteps;
    frameTime.base          = 0;
    frameTime.duration      = theNumberOfSteps;
    frameTime.rate          = 0;
    frameTime.recordSize    = sizeof(frameTime);
    frameTime.frameNumber   = 1;
    frameTime.flags         = icmFrameTimeHasVirtualStartTimeAndDuration;
    frameTime.virtualStartTime.lo   = 0;
    frameTime.virtualStartTime.hi   = 0;
    frameTime.virtualDuration       = theNumberOfSteps;

    HLock(myEffectDesc);
    DecompressSequenceFrameWhen(gEffectSequenceID,
                                StripAddress(*myEffectDesc),
                                GetHandleSize(myEffectDesc),
                                0,
                                0,
                                nil,
                                &frameTime);
    HUnlock(myEffectDesc);
}
```

The code in Listing 22-13 executes an effect in 30 steps.

Listing 22-13 Executing an effect directly by calling RunEffect

```
for (currentTime = 0; currentTime < 30; currentTime++)
{
    myErr = RunEffect(currentTime, 30);
    if (myErr != noErr)
        goto bail;
}
```

Creating New Video Effects

You should read this section if you want to write your own video effects. If you are only interested in building applications that use effects, you can skip this section.

QuickTime video effects are implemented as Component Manager components—the standard mechanism for extending QuickTime. To implement your own effect, you create a new **effect component**.

An effect component is a specialized type of image decompressor component. Chapter 4, “Image Compressor Components,” describes building image decompressor components.

The next section takes you through the implementation of a sample effect component. The sample effect is built on a framework of code that you can re-use when you implement your own effect component.

What Effects Components Do

The basic task of every effect component is very simple. The component is passed zero or more source frames and must produce a single destination frame. The destination frame is the source frame or frames after processing by the effect-rendering algorithm.

The component must provide a set of services that QuickTime can call. These services allow QuickTime (or any other client software that uses your component) to perform actions such as:

- Open a connection to your component.
- Retrieve information about your effect, particularly descriptions of the parameters your effect can take.
- Set the source or sources for the effect.
- Set the destination for the effect.
- Request that a single frame of the effect is rendered.
- Cancel the rendering of a frame.
- Close the connection to your component.

Your effect component must be able to service such requests. To do so, it implements a set of standard **interface functions** that are called through a **component dispatch** function. Details of these functions are given in the section “The Effect Component Interface” (page 640).

The main task of the effect component is to implement the specific algorithm that transforms source frames into a destination frame. You will need to supply versions of your algorithm for each bit depth and pixel format that your component supports. Choosing which bit depths and pixel formats to support, and implementing algorithms for each combination of these, is a significant part of building your effect component.

In addition, your effect component must provide a **parameter description** that describes the parameters that the effect takes. The parameter description can be used by the software that is calling your component to construct a user interface that allows users to change the parameters sent to your component. This is described in detail in the section "Supplying Parameter Description Information" (page 642).

The Effect Component Interface

Effect components, as with all other types of QuickTime components, must implement a defined set of functions. To ease the component development process, the `GenericEffect` component is provided for you. This component implements many of the "housekeeping" functions that all components must perform. In most cases, these default implementations are appropriate for your effect, and you simply delegate these functions to the generic effect component. In the rare instances when you need to provide your own implementations of one of these basic functions, you can override the generic version and provide your own implementation.

By delegating many of the functions to the generic effect, you not only decrease the number of functions you must implement, you also produce a smaller effect component, because common code is stored only once, in the generic effect.

The framework code provided in the dimmer effect sample (page 661) shows how to delegate interface functions to the generic effect component.

The remaining functions that your component must provide implementations for are:

- `Open` opens a connection between the client software and your component.
- `Close` closes the connection between the client and your component.
- `Version` returns the version number of your component.
- `EffectSetup` is called once before a sequence of frames are rendered. This gives your effect the chance to set up variables that will alter their value during the execution of a sequence of frames.
- `EffectBegin` is called once before a frame is rendered. Your component can safely perform operations that move memory when this function is called.

- `EffectRenderFrame` is called to render a frame. Because this function can be called asynchronously, it is not safe to perform operations that may move memory during this call.
- `EffectCancel` cancels the rendering of a frame. If your component supports asynchronous operation, this function can be called while a frame is being rendered.
- `GetParameterListHandle` returns a parameter description atom container, as described in the section "Supplying Parameter Description Information" (page 642).
- `GetCodecInfo` returns information to the codec manager about the capabilities of your component.
- `EffectGetSpeed` returns the approximate number of frames per second that your effect is capable of transforming.

These functions can be categorized into four groups. The `Open`, `Close` and `Target` functions deal with maintaining a connection between your component and client software. In most cases, you will not need to alter the implementations of these functions from the sample code provided by Apple.

The `Version`, `GetParameterListHandle`, `GetCodecInfo` and `EffectGetSpeed` functions return information about your component. The most important of these functions is `GetParameterListHandle`, which returns a description of the parameters that your effect can take. See "Supplying Parameter Description Information" (page 642) for more details of this what this function should do.

The `EffectSetup` function is called immediately before your component is required to render a sequence of frames. On entry, the function contains a description of the sequence that is about to be rendered. Most importantly, it describes the bit depth and pixel format of the sources that your component has to deal with. Your `Setup` function can then verify that your component can handle these formats. If it cannot, `EffectSetup` should return the "closest" bit depth and pixel format combination that it can handle, and QuickTime will generate versions of the sources and destination in the requested format. This ensures that your effect only ever handles source and destination buffers in a format it understands. See "The EffectRenderFrame Function" (page 655) for more details.

The most significant group contains the `EffectBegin`, `EffectRenderFrame` and `EffectCancel` functions. These functions contain the implementation of your effect algorithm. In most cases, you will not need to alter the implementations of the `EffectCancel` function from the sample code provided by Apple. The implementation of `EffectBegin` and `RenderFrame` functions is covered in "Implementing EffectBegin and EffectRenderFrame Functions" (page 647).

Full details of the interface functions your component must supply are given in "Component-Defined Functions" (page 740).

Supplying Parameter Description Information

Your effect must supply information to its client software that describes the parameters that the effect takes. Each parameter is described using a specific format, which is described in "The Parameter Description Format" (page 668).

The easiest way to supply this information back to the client software is to add an `'atms'` resource to your component. The `'atms'` resource contains the parameter descriptions in the required format. You can retrieve the resource by calling the `GetComponentResource` function, as shown in Listing 22-14.

Listing 22-14 An implementation of the GetParameterListHandle function

```
pascal ComponentResult GetParameterListHandle(EffectGlobals *glob,
                                                Handle *theHandle)
{
    OSErr   err = noErr;

    err = GetComponentResource((Component) glob->self,
                        OSTypeConst('atms'),
                        kEffectatmsRes,
                        theHandle);

    return err;
}
```

By implementing the `GetParameterListHandle` function in this way, you reduce the problem to supplying an appropriate `'atms'` resource with your component.

Adding an 'atms' Resource to your Component

The `'atms'` resource for your effect contains two sets of information. The first set contains the "standard information" that is used to construct the standard parameters dialog box. This includes items such as the title of the dialog box, and optional copyright information.

The second set contains the "parameter information," which is a description of each parameter that your effect takes. If your effect does not take parameters, there will be no information in this set.

The structure of an `'atms'` resource is as follows. The header for this resource contains the resource ID and the number of root level atoms it contains:

```
resource 'atms' (kEffectatmsRes) {
7,
{
    // The resource body goes here
};
};
```

The first line contains the ID of the `'atms'` resource. In this example, the identifier that is used (`kEffectatmsRes`) is also used in the call to `GetComponentResource` in Listing 22-14. This ensures that the right `'atms'` resource is read by QuickTime.

The second line contains the number of root atoms in the resource. Each `'atms'` resource contains a number of atoms. The number in the second line must contain a count of the number of first-level atoms in the resource.

▲ **WARNING**
It is important that you ensure this number is updated if you add and delete atoms from your `'atms'` resource. If this number is smaller than the number of atoms actually in the resource, you will not be able to adjust the values of all the parameters of the effect. If the number is larger than the number of atoms in your effect, your effect component may cause QuickTime to crash when it is used. ◆

The body of the `'atms'` resource consists of a number of atom declarations. Each declaration has a header that contains:

- the atom name
- the atom ID
- the number of children in the declaration

The header is followed by the atom's data, which is either one or more typed values, such as a `string` or a `long`, or a set of child atoms.

Listing 22-15 shows an example atom that contains a single typed value as its data. Note that the value is a type followed by the data itself. The number of children of the atom is declared as `noChildren` because the atom contains a typed value.

Listing 22-15 An example 'atms' atom declaration

```
kParameterTitleName, kParameterTitleID, noChildren,
{
    string { "Dimmer2 Effect Parameters" };
};
```

The Standard Information in an 'atms' Resource

The Standard Information stored in an `'atms'` resource is made up of three required atoms and five optional atoms.

The three required atoms are:

- `kParameterTitleName` - a `string` that is used as the title of the standard parameters dialog box. An example of a `kParameterTitleName` atom declaration is shown in Listing 22-15.
- `kParameterWhatName` - an `OSType` containing the name of this effect component.
- `kParameterSourceCountName` - a long integer containing the maximum number of sources that the effect can take.

The five optional atoms are:

- `kParameterAlternateCodecName` - an `OSType` containing the unique identifier of another effect component that should be used to replace this effect if this effect cannot be used.
- `kParameterInfoLongName` - a `string` containing the long version of the name of the effect
- `kParameterInfoCopyright` - a `string` containing a copyright statement for the effect.
- `kParameterInfoDescription` - a `string` containing a brief description of what the effect does.

- `kParameterInfoWindowTitle` - a `string` containing the title of the window that displays the information contained in the optional atoms.

The Parameter Information in an 'atms' Resource

For each parameter of the effect your `'atms'` resource must contain a set of atoms in the `'atms'` resource that describes that parameter. This description includes the name of the parameters, the type and range of values it can take and hints on appropriate user interface that client applications should present to allow users to edit this parameter.

A complete description of the information you need to provide for each parameter can be found in "The Parameter Description Format" (page 668).

For a basic parameter, there are five atoms that you should supply:

- `kParameterAtomTypeAndID` - contains the type and ID of the parameter (an `OSType` and a long integer, respectively), the atom flags for the parameter and a `string` containing the name of the parameter.
- `kParameterDataType` - a long integer containing the type of the parameter.
- `kParameterDataRange` - a set of typed values describing the range of values the parameter can take. The exact values you supply depend on the value of the `kParameterDataType` atom.
- `kParameterDataBehavior` - two long integer values containing the behavior type and flags
- `kParameterDataDefaultItem` - the default value of the parameter. Again, the type of this value will depend on the type of the `kParameterDataType` atom.

An example of a basic parameter description is shown in Listing 22-16.

Listing 22-16 An example set of parameter description atoms

```
kParameterAtomTypeAndID, 101, noChildren,
{
    OSType { "sden" };      // atomType - the name of this parameter
    long { "1" };           // atomID - this is atom number 1
    kAtomNotInterpolated; // atomFlags - this parameter cannot be tweened
    string { "Scratch Density" }; // atomName - the name of the parameter
            // as it will appear in the standard parameters dialog box
};
```

```
kParameterDataType, 101, noChildren,
{
    kParameterTypeDataLong;     // dataType - this parameter contains a
                                // long value
};

kParameterDataRange, 101, noChildren,
{
    long { "0" };       // Minimum value
    long { "25" };      // Maximum value
    long { "1" };       // Scale factor - no scaling is applied to this
                        // parameter
    long { "0" };       // Precision - 0 indicates that this parameter is
                        // not a floating point value
};

kParameterDataBehavior, 101, noChildren,
{
    kParameterItemControl;  // behavior type - this parameters should be
                            // represented by a slider
    long { "0" };           // behaviorFlags - no flags
};

kParameterDataDefaultItem, 101, noChildren,
{
    long { "5" };           // The default value of the parameter
};
```

Tweening Parameters

An important property of effect parameters is that they can be tweened. Tweening is QuickTime’s general purpose interpolation mechanism. For many parameters, it is desirable to allow the value of the parameter to change as the effect executes.

For example, the slide effect built into QuickTime (see “Slide ('slid')” (page 715)) has an `angle` parameter. This controls the angle from which the second source will slide over the first during the execution of the effect. If this parameter contains a single value, the second source will slide in over the first in a straight line. However, if the parameter contains two values, the angle will be interpolated between these values during the execution of the effect. This allows you to specify a curved slide effect.

In fact, any valid tween record can be specified as the parameter value, not just records containing pairs of values. The QuickTime tweening mechanism supports tween records that contain more than two values and that specify the interpolation algorithm used to produce intermediate values. However, the standard parameters dialog box only allows a pair of values to be entered, and the appropriate default interpolator is used. An application can provide its own user interface for entering multiple tween values for a parameter and choosing an appropriate tweener to perform interpolation, if required.

Note
Effect component authors do not need to write code to handle all the possible combinations of tween record types as the details of the tween record are handled by using the standard QuickTime tweening APIs. For a complete description of tweening, see Chapter 25, "Tween Components and Native Tween Types." ◆

For more details on specifying which parameter values can contain tween values, see "Parameter Atom Type and ID" (page 668). For more details on supporting tweened parameters in your effect component, see "Tweening Parameter Values" (page 653).

Refer to "The Parameter Description Format" beginning on page 668 for a complete description of the possible parameter descriptions you can place in your `'atms'` resource.

Implementing EffectBegin and EffectRenderFrame Functions

The core of implementing an effect component is implementing the `EffectBegin` and `EffectRenderFrame` functions. Together, these functions handle the rendering of a single frame of the effect.

The `EffectBegin` function is called immediately before each frame is to be rendered. It is guaranteed that this function is never called from an interrupt, so it is safe to perform actions that could move memory within this function. In general, the `EffectBegin` function should set up the internal state of your component so it has all the information it needs to render a single frame.

The `EffectRenderFrame` function is called to actually render the frame. This can be called at interrupt time, so it is not safe to move or allocate memory in this function. You should also take care not to call functions that would do so. Your `EffectRenderFrame` function should actually render a single frame of your effect.

The EffectBegin function

The main tasks that the `EffectBegin` function should perform are:

- Ensuring that the effect component has valid references to the current sources. If the component does not have a reference to the sources, or the sources have changed since the last call to `EffectBegin`, they must be updated.
- Ensuring that the component has a valid reference to the current destination. If the component does not have a reference to the destination, or the destination has changed since the last call to `EffectBegin`, it must be updated.
- Ensuring that the component has the current parameter values. If the source or destination has changed, or the component does not currently have values for the effect parameters, these parameter values are read.
- If any of the parameter values are tweened, tweening is performed to determine the actual value for those parameters.

Checking Source and Destination References

The following code checks to see if the destination has changed since the last call to the `EffectBegin` function:

```
if (p->conditionFlags & (codecConditionNewClut+
                    codecConditionFirstFrame+codecConditionNewDepth+

    codecConditionNewDestination+codecConditionNewTransform))
```

If this evaluates to `true`, the destination has changed. This expression checks a series of flags that are passed to the `EffectBegin` function in the `conditionsFlags` field of the `decompressParams` parameter. When the destination is changed, QuickTime sets these flags to alert the effect component to update its internal state.

The most important information that you need to store about the new destination is its base address and its `rowBytes` value. These values allow you to draw onto the destination surface.

Listing 22-17 shows an example function that stores information in the effect component's global data structure about the destination `PixMap` passed to the function.

Listing 22-17 Storing information about a new destination frame

```
static long BlitterSetDest(BlitGlobals*glob,    // input: our globals
    PixMap  *dstPixMap,     // input: pixels we will draw into
    Rect    *dstRect)       // input: area of pixels we will draw into
{
    OSErr   result = noErr;
    long    offsetH,offsetV;
    char    *baseAddr;

    // Calculate the based address according to the format of the
    // destination PixMap
    offsetH = (dstRect->left - dstPixMap->bounds.left);
    if (dstPixMap->pixelSize == 16)
    {
        offsetH <<= 1;          // 1 pixel = 2 bytes
    }
    else
    {
        if (dstPixMap->pixelSize == 32)
        {
            offsetH <<= 2;      // 1 pixel = 4 bytes
        }
        else
        {
            result = -1;        // this is a data format we can't handle
        }
    }
    offsetV = (dstRect->top - dstPixMap->bounds.top)
                * dstPixMap->rowBytes;
    baseAddr = dstPixMap->baseAddr + offsetH + offsetV;

    glob->dstBaseAddr = baseAddr;
    glob->dstRowBytes = dstPixMap->rowBytes;
    return result;
} // BlitterSetDest
```

The process for checking for new sources is broadly similar. The `CodecDecompressParams` data structure passed into the `EffectBegin` function has a field called `majorSourceChangeSeed`. This contains a seed number generated from the characteristics of the set of sources for the effect. If the sources change, the `majorSourceChangeSeed` value will also change, so the effect can store the current value in its global data structure and compare it to the current value. If they are different, the effect knows its sources have changed.

When the effect detects that one or more of its sources have changed, it must iterate through all its sources and reload information about them.

Listing 22-18 shows example code that performs these operations. Listing 22-19 shows the `BlitterSetSource` function that is called by this example code. The `BlitterSetSource` function is analogous to the `BlitterSetDest` function shown in Listing 22-17.

Listing 22-18 Checking for source changes

```
// Check to see if one or more sources have changed
if (p->majorSourceChangeSeed != glob->majorSourceChangeSeed)
{
    // grab start of input chain for this effect
    source = effect->source;

    // we can play with up to kMaxSources sources, so go get them
    while (source != nil && numSources < kMaxSources)
    {
        // now give that source to our blitter
        err = BlitterSetSource(glob, numSources, source);
        if (err != noErr)
            goto bail;

        source = source->next;
        ++numSources;
    }
}
```

Listing 22-19 Storing information about a new source frame

```
static long BlitterSetSource(BlitGlobals*glob,      // input: our globals
    long sourceNumber,                    // input: source index to set
    CDSequenceDataSourcePtr source)     // input: source value
{
    OSErr   err = noErr;

    if (sourceNumber >= kMaxSources)
    {
        // too many sources for us to handle
        return noErr;
    }
    else
    {
        // a source we can handle, save it away
        err = RequestImageFormat(source, glob->width, glob->height,
                                    glob->dstPixelFormat);
        if (err == noErr)
        {
            glob->sources[sourceNumber].src = source;
        }
        else
        {
            glob->sources[sourceNumber].src = nil;
        }

    }
    return (err);

} // BlitterSetSource
```

Reading Parameter Values

Listing 22-20 shows how to read the value of a non-tweened parameter. The `QTFindChildByID` function is used to retrieve the atom containing the parameter value. The parameter value is then copied from the atom using the function `QTCopyAtomDataToPtr`. If the value is successfully copied, it is endian-flipped to ensure it is in native-endian format (parameter values are always stored in big-endian format). If the copy failed, a default value is provided.

The value retrieved from the parameter is stored in the component's global data structure (called, in this example, `global->blitter`). This allows the value to be used by other functions, notably the component's `EffectRenderFrame` function.

Listing 22-20 Reading a parameter value

```
{
    Ptr             data = p->data;
    QTAtom          atom;
    QTAtomID        atomID = 1;
    long            actSize;

    // Find the 'sden' atom
    atom = QTFindChildByID((QTAtomContainer) &data,
                        kParentAtomIsContainer,
                        OSTypeConst('sden'), // The name of the parameter
                        atomID,              // The ID of the parameter
                        nil);

    // Copy the parameter value from the atom
    if (QTCopyAtomDataToPtr((QTAtomContainer) &data,
                        atom,
                        false,
                        sizeof(long),
                        &((glob->blitter).scratchDensity),
                        &actSize)!=noErr)
    {
        // If the copy failed, use a default value for this parameter
        ((glob->blitter).scratchDensity) = 1;
    }
    else
    {
        // Otherwise, the copy succeeded, so endian flip and store the
        // parameter value
        ((glob->blitter).scratchDensity) =
EndianS32_BtoN(((glob->blitter).scratchDensity));
    }
}
```

If the parameter value can contain a tweened value, you can use code similar to that shown in Listing 22-21 to retrieve the parameter value. The functions `InitializeTweenGlobals` and `CreateTweenRecord` are utility functions that Apple provides as part of the dimmer effect sample framework (see "The Sample Effect Component" (page 661)).

Listing 22-21 Reading a tweened parameter value

```
{
    Ptr                 data = p->data;
    OSErr               err;
    long                index = 1;

    err = InitializeTweenGlobals(&glob->tweenGlobals, p);
    if (err!=noErr)
        goto bail;

    // Make our tweener, return if we already have it
    err = CreateTweenRecord(&glob->tweenGlobals,
                    &glob->percentage,
                    OSTypeConst('pcnt'),    // The name of the parameter
                    1,                      // The ID of the parameter
                    sizeof(Fixed),
                    kTweenTypeFixed,
                    (void*) 0,
                    (void*) fixed1,
                    effect->frameTime.virtualDuration);
    if (err!=noErr)
        goto bail;

    glob->initialized = true;
}
```

Tweening Parameter Values

If you have specified that one or more of your parameter's values can be tweened, you need to implement code to perform the tweening in the `EffectBegin` function.

Listing 22-22 shows an example of tweening a parameter value. The current frame time is retrieved and subtracted from the effect's `virtualStartTime`. This calculates how far through the execution of the current effect sequence we are, expressed as a percentage.

With this information, the code then calls `QTDoTween` to interpolate the parameter value, leaving the resulting value in `glob->comp1Tween.tweenData`.

Listing 22-22 Tweening parameter values

```
wide    percentage;

// Find out how far through the effect we are
percentage = effect->frameTime.value;
CompSub(&effect->frameTime.virtualStartTime, &percentage);

// Tween our parameters and get the current value for this frame, prepare
// to render it when the EffectRenderFrame happens
{
    Fixed   thePercentage;

    if (glob->percentage.tween)
        QTDoTween(glob->percentage.tween, percentage.lo,
                        glob->percentage.tweenData, nil, nil, nil);

    thePercentage = **(Fixed**) (glob->percentage.tweenData);

    // If we are before the half-way point of this transition, we should
    // be fading the first source to black
    if (thePercentage < fixed1/2)
    {
        (glob->blitter).direction = 1;
        (glob->blitter).dimValue = FixedToInt(FixMul(IntToFixed(512),
thePercentage));
    }
    // Otherwise, we are fading up onto the new source
    else
    {
        (glob->blitter).direction = 0;
        (glob->blitter).dimValue = FixedToInt(FixMul(IntToFixed(512),
                                                  thePercentage)) - 255;
    }
}
```

The EffectRenderFrame Function

The `EffectRenderFrame` function is called to actually render a single frame of your effect. This is where you transform the sources of your effect into the destination frame, using the algorithm that implements your effect.

This is also where you have to handle multiple bit depth and pixel format combinations.

Internally, QuickTime stores bitmaps in a wide variety of formats. The system can handle images in a number of bit depths and with many different pixel formats. Effect components must have some ability to handle source and destination frames that are at any of the bit depths and in any of the pixel formats that QuickTime supports.

Obviously, providing a separate implementation of your effect algorithm for every combination of bit depth and pixel format could be an enormous task. Fortunately, QuickTime provides mechanisms for you to limit the number of formats you have to explicitly support.

When your effect component's `EffectSetup` function is called, it is passed information about the bit depth and pixel formats that the source frames are in. Your component should examine the formats and react in one of two ways:

■ If the format is one of those which your effect does support, the `EffectSetup` function does nothing.

■ If the format is not supported by your effect, `EffectSetup` returns the nearest format that is supported.

In the second case, where you do not directly support the format, QuickTime automatically creates buffers in the format returned by `EffectSetup`. The source frames are written into the buffer before `EffectRenderFrame` is called, so that source data is always available in a supported format. The destination frame is also buffered, and QuickTime automatically transforms the image into the required format for you.

This way, you only need to support a limited number of image formats, and QuickTime will ensure that `EffectRenderFrame` isn't called with data in any other format.

Note
If your effect does not handle the bit depth and pixel format combination passed to the `Setup` function, and it requests an alternative format, QuickTime generates new offscreen buffers for each source and destination frame your effect uses. This will result in a memory and execution time overhead for your effect. If you want your effect to execute quickly in a wide range of circumstances, your effect should explicitly handle as many bit depth and pixel format combinations as possible. ◆

Handling Multiple Formats

Although you can write separate versions of your effect algorithm for each combination of bit depth and pixel format, Apple recommends that you implement your effect algorithm once for each bit depth. You should then use the Apple-supplied **blit macros** to automatically generate versions of these implementations for each supported pixel format. This significantly reduces the number of separate implementations you have to maintain, and allows easy support of multiple pixel formats.

The blit macros are contained in the file `BltMacros.h`, which is included with the sample effect framework code.

To use the blit macros in your effect component, you must store each bit depth implementation of your effect algorithm in a separate file. These files are then included into the effect component's main source code file multiple times, once per pixel format supported. Each inclusion is surrounded by `#define` statements that define the pixel format version to be generated.

Each file uses a `#include` statement to include `BltMacros.h`, and all operations that read pixels from a source buffer or write pixels to the destination buffer are performed using appropriate macros.

The macros are automatically converted to the correct operations for the pixel format when the file is included into the main source code. This generates a version of the algorithm for each pixel format.

Finally, code is put into place in the effect component's `EffectRenderFrame` function, which calls the appropriate generated algorithm according to the current bit depth and pixel format of the source buffers.

Implementing a Bit-depth Specific Version of Your Algorithm

Listing 22-23 shows an example implementation of an effect algorithm. The code uses the blit macros to read pixels from the source frame and write them to the destination frame. This example shows a filter that changes a single source. It also shows how to read and alter a single pixel at a time; other effects may handle multiple pixels at a time for efficiency.

The sample shows the following operations for each pixel of the source frame:

1. Retrieving the next pixel from the source, using the `Get16` (which reads a 16-bit pixel from a memory address) and `cnv16SPFto16RG` (which converts a 16-bit pixel in the current pixel format to the standardized 16-bit ARGB format) macros to handle pixel format conversion;
2. Decomposing the pixel into alpha, red, green and blue components;
3. Reassembling the alpha, red, green and blue components into a standardized ARGB pixel value;
4. Writing the pixel value to the destination buffer, using the `cnv16RGto16DPF` (which converts the 16-bit standardized format pixel back into the current buffer's 16-bit pixel format) and `Set16` (which writes a 16-bit pixel to a memory address) macros to handle pixel format conversion.

The actual effect implementation, which would alter the alpha, red, green and blue values of each pixel according to the effect specification, is not shown in this example code.

Listing 22-23 A sample effect algorithm for 16-bit frames

```
#include <BltMacros.h>
void EffectFilter16(BlitGlobals *glob);
void EffectFilter16(BlitGlobals *glob)
{
    long    height = glob->height;      // Local copy of the height of
                                        // the buffers
    UInt16  *srcA = glob->sources[0].srcBaseAddr;   // Local pointer to
                                                    // the first source image
```

```
UInt16  *dst = glob->dstBaseAddr;   // Local pointer to the
                                    // destination
long    srcABump;
long    dstBump;

// Work out the source and destination "bumps". The rowBytes value
// gives you the number of bytes in each scanline of an image. This
// is not necessarily the same as the number of pixels in a scanline
// multiplied by the number of bytes each pixel occupies. When
// we copy pixels from source to destination, via our effect
// algorithm, we need to account for this discrepancy. The following
// lines lines pre-calculate the differences.
srcABump = glob->sources[0].srcRowBytes - (glob->width * 2);
dstBump  = glob->dstRowBytes - (glob->width * 2);

// Now, for every scanline in the source image we are dealing with...
while (height--)
{
    long    width = glob->width;

    // ...iterate through every pixel in that scanline
    while (width--)
    {
        UInt16      thePixelValue;

        // Retrieve the next pixel value
        thePixelValue = Get16(srcA);
        srcA++;

        // Call to blit macros to ensure the pixel format is
        // converted appropriately
        cnv16SPFto16RG(thePixelValue);

        // Get the alpha, red, green and blue values of the pixel
        alpha = 0x8000 & thePixelValue;
        red   = (thePixelValue & 0x7C00) >> 10;
        green = (thePixelValue & 0x03E0) >> 5;
        blue = (thePixelValue & 0x001F) >> 0;

        // IMPLEMENT YOUR EFFECT ALGORITHM HERE ON EACH PIXEL
```

```
            // Re-assemble the A, R, G and B values into a 16-bit
            // destination pixel
            thePixelValue = alpha | (red << 10) | (green << 5)
                                  | (blue << 0));

            // Set the destination pixel,first passing it through the
            // appropriate blit macro to
            // ensure the correct pixel format conversion is performed
            cnv16RGto16DPF(thePixelValue);
            Set16(dst, thePixelValue);
            dst++;
        }

        // Bump the source and destination pointers we are using, to
        // avoid problems when moving from one scanline to the next
        srcA = (void *) (((Ptr) srcA) + srcABump);
        dst = (void *) (((Ptr) dst) + dstBump);
    }
}
```

Including the Bit-depth Implementations into Your Effect Code

Once you have produced separate implementations of your effect algorithm for each bit depth you support, you need to include these into your main effect source code file. Each bit depth implementation is included once for every pixel format you support.

Listing 22-24 shows statements to include the 16-bit implementation of the effect into the main effect source code file. The implementation is included three times, for the following pixel formats:

1. Big-endian 555 RGB
2. Little-endian 555 RGB
3. Little-endian 565 RGB

The result of the code in Listing 22-24 is that your effect source code contains three separate versions of the effect algorithm for handling 16-bit sources. These are named `EffectFilter16BE555`, `EffectFilter16LE555` and `EffectFilter16LE565`, respectively.

Listing 22-24 Including the 16-bit implementation into the main effect source code

```
// 16-bit, Big Endian 555 pixel format
#define EffectFilter16 EffectFilter16BE555
#define srcIs16BE555 1
#define dstIs16BE555 1
#include "EffectFilter16.c"
#undef EffectFilter16
#undef srcIs16BE555
#undef dstIs16BE555

// 16-bit, Little Endian, 555 pixel format
#define EffectFilter16 EffectFilter16LE555
#define srcIs16LE555 1
#define dstIs16LE555 1
#include "EffectFilter16.c"
#undef EffectFilter16
#undef srcIs16LE555
#undef dstIs16LE555

// 16-bit, Little Endian, 565 pixel format
#define EffectFilter16 EffectFilter16LE565
#define srcIs16LE565 1
#define dstIs16LE565 1
#include "EffectFilter16.c"
#undef EffectFilter16
#undef srcIs16LE565
#undef dstIs16LE565
```

Calling the Effect Implementations from EffectRenderFrame

Finally, you must provide code inside your `EffectRenderFrame` function to call the appropriate implementation of your effect algorithm, depending on the pixel format and bit depth of the source frames you are dealing with. Listing 22-25 shows how to do this for the 16-bit pixel formats.

Listing 22-25 Calling pixel format specific versions of the 16-bit effect implementation

```
switch (glob->dstPixelFormat)
{
    case k16BE555PixelFormat:
            EffectFilter16BE555(glob);
            break;
    case k16LE565PixelFormat:
            EffectFilter16LE565(glob);
            break;
    case k16LE555PixelFormat:
            EffectFilter16LE555(glob);
            break;
}
```

The code to handle the 32-bit pixel formats is an easy extension of the code shown in this section, and can be found in the sample effect component included in the QuickTime 3 SDK and described in detail in the next section.

The Sample Effect Component

This section introduces you to the sample effect component supplied as part of the QuickTime 3 SDK. It takes you through the parts of the code that you will need to change in order to implement your own effect component.

Introduction to the Dimmer Effect

The sample effect described in this section is a dimmer effect. This simple effect fades the first source to black, then fades up to show the second source. The source code for this effect is provided as a CodeWarrior Pro project as part of the QuickTime 3 SDK.

The Standard Effect Framework

Much of the code required to implement an effect component is the same for all components. Apple has provided a framework of code that you can adapt to create your own effect components. In most cases, you only have to change limited portions of the framework code to create your new component.

For more detailed information on writing components, see Chapter 9 of *Mac OS For QuickTime Programmers*; and Chapter 4, “Image Compressor Components,” in this reference.

Structure of the Framework

The effect framework is one approach to writing effects components. It has been designed to provide most of the basic code required to implement an effect component, leaving the implementation of the effect algorithm to you. It should be possible to write most effects using the framework, though you may need to adapt the framework code for more complex effect components.

The framework implements the required component functions for an effect. The main body of the framework is the `Effect.c` file, which contains the source code for the framework and an implementation of the dimmer effect. This file is made up of four main parts:

- The **global data structures** used by the framework. You will need to update some of the data structures to reflect the capabilities of the effect you are implementing.
- The **dispatcher** is the entry point to your component. Because all effects components have the same set of component functions, you should not need to alter the dispatcher.
- The **internal functions** are the set of functions that actually execute your effect. This is where most of your own code will be added.
- The **component functions** are the standard functions called by the dispatcher. These functions call the internal functions to actually execute the effect. For most effects, you won’t need to change much code in this section.

Naming Conventions

All the function and data structure names in the framework are arbitrary. The names have been chosen to reflect their purpose, but you are free to change the names, as long as they remain internally consistent.

If you choose to change the names of the component functions, you will have to change the `CALLCOMPONENT_BASENAME #define` in `Effect.c`. This defines the root of the name used for component functions. For example, if `CALLCOMPONENT_BASENAME` is set to `SlideEffect`, then the Open component function must be called `SlideEffectOpen`, the Close component function must be called `SlideEffectClose`, and so forth.

Apple recommends that you do not change the names used in the framework.

Writing an Effect Component Using the Framework

The effect component framework is provided for you to simplify the development of QuickTime video effects components.

The QuickTime 3 SDK includes the folder `DimmerEffect`, which contains the framework, complete with associated resources and makefiles. See the `ReadMe` file in the `DimmerEffect` folder for full installation and use instructions.

To adapt the dimmer framework to create your own component, search through the source code file `Effect.c` for the comments “`*** CHANGE ***`”. These comments mark the sections of the source code you will need to change to write your own effect.

The following sections take you through the specific changes you need to make to the framework. All the changes except the last, which implements the actual effect algorithm, are made to the `Effect.c` file.

Synchronous vs. Asynchronous Processing

The first change you may need to make is to the following `#define`:

```
#define kMaxAsyncFrames 0
```

This value defines the number of frames that can be queued for asynchronous rendering by this effect. If your effect declares that is can handle more than `0` asynchronous frames, frames may be queued for rendering. If you wish to render synchronously, set `kMaxAsyncFrames` to `0`, otherwise, set it to the number of frames that can be held in the queue.

Defining the Number of Sources

Most effects require one or more sources to operate on, though some effects—such as Apple’s fire effect—operate without any sources. The dimmer effect uses two sources: the first is the source to fade to black, the second is the source to fade up on. Most effects transition between two sources. Sometimes, effects control the transition between two scenes by blending in one or more other sources, in which case the effect may require three or more sources. You may also want to implement a filter effect that has only a single source and produces a transformed version of that source.

You set the value of `kMaxSources` to the maximum number of sources required by the effect. Effects that can take more than one source should be prepared to handle the case when fewer than the maximum number of sources are actually provided. For example, if your effect expects two sources to transition between and a third source to use as a mask, your code must handle the case where only the two transition sources are provided. In this case you should use a default mask instead of a third source.

Adding to the Global Data Structures

The framework defines two global data structures: `BlitGlobals` and `EffectGlobals`. The `BlitGlobals` structure holds information related to drawing a single frame of the effect, while the `EffectGlobals` holds data for the entire effect as it is executed. These data structures are global to an instance of the effect component. That is, if you have multiple instances of the component opened, each instance gets its own copy of both data structures.

You can add fields to the `BlitGlobals` data structure to hold information specific to your effect. A set of standard fields are already defined, which hold information used by the framework. You can add your own effect-specific fields between the `*** CHANGE ***` and `*** END CHANGE ***` comments.

The example defines two fields, `dimValue` and `direction`. These hold the current dim value for the effect and a flag indicating whether it is fading down or up, respectively. Because the dimmer fades the first scene down to black then fades up on the second scene, it also needs to store the dim value between individual frames. This value is stored in the `dimValue` field of `BlitGlobals`.

You can also add fields to the `EffectGlobals` structure. Generally, you will read the values for the parameters to your effect in these fields so that they can be referenced while the effect executes.

Preflighting the Blitter

The internal function `BlitterPreflight` is called from `EffectSetup` before the first frame of the effect is rendered. This function’s main task is to validate the bit depth that the effect is being requested to support.

The bit depth that the effect is being asked to operate at is passed in the `depth` parameter to `BlitterPreflight`. The function should return in the same parameter the bit depth at which it wants to operate.

For example, the dimmer effect can operate on 16-bit or 32-bit sources. If either of these values is passed in, it simply returns `depth` unaltered. If any other bit depth is requested, it sets `depth` to `16`, the default bit depth for this effect.

Your effect should validate the bit depth passed in a similar way. Apple recommends that your effect support at least 16- and 32-bit depths.

When you set the `depth` parameter to a different value than it was on entry to `BlitterPreflight`, QuickTime creates an offscreen buffer for the sources and destination of the effect. All data is passed through these offscreen buffers, to ensure that your effect only sees data in a format it can handle.

Setting the Destination

The `BlitterSetDest` function is called from `EffectBegin` and is passed the effect's destination, in the form of a `PixMap`. The `BlitterSetDest` function should calculate the base address and rowBytes values for the destination and store these in the `BlitGlobals` data structure for future reference.

You need to make changes to this function only if your effect supports destinations in bit depths other than 16-bit and 32-bit.

The BlitterRenderFrame function

This function calls the functions that implement your effect algorithm. The function names to be called are those generated by the blit macros.

The example code supports the three most common pixel formats in 16-bit and 32-bit. If your effect needs to support other bit depths or pixel formats, you need to update the `switch` statement in this function so that the appropriate drawing functions are called.

The EffectsFrameClose function

This function is called when the client software has finished using your component. At this time, your component should dispose of any memory it allocated. In particular, you should call `DisposeTweenRecord` for each tween record you allocated and then call `DisposeTweenGlobals`.

Reading the Effect Parameters

The parameters of the effect are read in the `EffectsFrameEffectBegin` function. Your effect should read its parameter values in the section between the

`*** CHANGE ***` and `*** END CHANGE ***` comments, reading either non-tweened or (more frequently) tweened values. Example code for both these cases is given in “Reading Parameter Values” (page 651).

Once you have read in the parameter values, you need to tween those parameters that contain tween records. This code should be placed between the second pair of `*** CHANGE ***` and `*** END CHANGE ***` comments. Again, example code to do this is supplied, see “Tweening Parameter Values” (page 653).

Implementing your Effect

The last stage in adapting the framework is to implement your effect algorithm. You need to provide one implementation per bit depth that your effect explicitly supports, and each implementation must be placed in a separate file. These files are named `EffectFilter16.c`, `EffectFilter32.c`, and so forth.

The dimmer effect code provides an example of the pixel manipulations that an effect will typically perform, and shows how to use the blit macros to support multiple pixel formats at a given bit depth.

Clearly, the details of these routines are entirely dependent on the effect being implemented.

Parameter Descriptions

Each effect component supplies a **parameter description** data structure that describes in detail the set of parameters that the effect has.

This section describes the parameter description format in detail. You need this information if you are writing an effect component and need to provide a parameter description as part of an `'atms'` resource (see “Supplying Parameter Description Information” (page 642) for more details). You may also need this information if you are writing an application that needs to present its own user interface for setting effect parameters. In this case, you will need to parse parameter descriptions to generate appropriate controls to set parameter values.

Any software that uses an effect can request its parameter description. Typically, the parameter description is then passed to `QTCreateStandardParameterDialog` or especially, `ImageCodecCreateStandardParameterDialog`. These functions use the parameter description to display a user interface that allows users to choose the values of the parameters.

Your applications are free to use the information in an effect's parameter description in other ways. For example, if you do not want to use the standard parameter dialog box, you can use the parameter description to assemble your own user interface for setting parameter values.

Parameter descriptions are stored as a `QTAtomContainer`, and you retrieve an effect's description by calling `ImageCodecGetParameterList`. This function takes a component instance and returns the parameter description for that component.

The code shown in Listing 22-26 opens the component specified in the variable `subType`. The code sets up the component description, then finds and opens the requested component. It then calls the `ImageCodecGetParameterList` function to fill out the parameter description for this effect.

Listing 22-26 Opening the image decompressor component

```
{
    // Set up a component description
    cd.componentType    = 'imdc';       // Effects are image decompressor
                                        // components
    cd.componentSubType = subType;      // This is the name of the effect
                                        //(e.g. 'smpt')
    cd.componentManufacturer    = 0;
    cd.componentFlags           = 0;
    cd.componentFlagsMask       = 0;

    // Find the required component. If it can't be found, generate an
    // error
    if ((theComponent = FindNextComponent(theComponent, &cd))==0)
    {
        err = paramErr;
        goto bail;
    }

    // Open the component
    gCompInstance = OpenComponent(theComponent);
    // Get the parameter description for the effect
    ImageCodecGetParameterList(gCompInstance, &parameterDescription);
}
```

Your application can parse the returned parameter description using the standard QuickTime APIs that query `QTAtomContainer` data structures. This can be useful if your application deals with effect components whose parameter list is not known ahead of time. In particular, if your application maintains its own user interface for users to customize effects, it will need to read and understand parameter descriptions.

This section describes the general format of the data returned in a parameter description.

The Parameter Description Format

The parameter description data structure is a `QTAtomContainer` that, when filled out by the `ImageCodecGetParameterList` call, contains a set of `QTAtoms` for each parameter of the effect. These atoms define the base type of the parameter, the legal range of values that can be stored in it, and hints for displaying a user interface that allows users to choose values for the parameter.

The atoms in a parameter description are described in the following sections. The order in which the atoms are stored in the `QTAtomContainer` is important. Your application should try to present parameters to the user in the same order that they are contained in the parameter description.

Each of the atoms in a parameter description has a name; you will find constants for these in `ImageCodec.h`. You should use these constants when retrieving atoms from the data structure. The data stored in the atoms of the parameter description is structured, and the `struct` definitions are given in the atom descriptions below.

Many of the atoms must be present to create a valid parameter description. Some, though are optional, as noted.

Parameter Atom Type and ID

This atom contains information about the type and ID of the parameter. The data is contained in the following structure:

```
typedef struct
{
    QTAtomType  atomType;
    QTAtomID    atomID;
    long        atomFlags;
    Str255      atomName;
} ParameterAtomTypeAndID;
```

`atomType`	This field contains either a unique identifier for the parameter or the value `kNoAtom`. The unique identifier is a four character `OSType` that you use to retrieve the parameter's value. If this field contains `kNoAtom`, the "parameter" being described is actually a group description; groups are described in"Special Description Types" (page 678).
`atomID`	This field contains the ID of this parameter.
`atomFlags`	This field can contain one of the predefined values: `kAtomNoFlags`, `kAtomNotInterpolated`, or `kAtomInterpolateIsOptional`. If it contains `kAtomNotInterpolated`, the user interface for allowing the user to specify parameter values only allows users to enter a single value for this parameter, and this value remains constant while the effect is playing. If it contains `kAtomNoFlags`, the user interface allows users to enter a set of values for the parameter. This set of values are stored in a tween atom, and the value of the parameter is interpolated between these values during effect playback. If it contains `kAtomInterpolateIsOptional`, the user interface defaults to allowing a single value for the parameter. If the user interface supports an "advanced" mode of operation, then a tween value can be entered for this parameter when the user interface is in this mode. An example of an advanced mode is the standard parameters dialog box: if you hold down the option key while selecting an effect, any parameters that are have the `kAtomInterpolateIsOptional` flag set, will allow a tween value to be entered.
`atomName`	The name of this parameter. This `string` value is used as the name of the control displayed in the standard parameter dialog box to enter a value for this parameter. This atom is required.

Parameter Data Type

This atom defines the type of the data for this parameter. It contains data in the following structure:

```
typedef struct
{
    OSType  dataType;
}
```

`dataType`	This field contains the type of the value that is stored in this parameter. This can be one of the following values:
	`kParameterTypeDataText` - editable text item
	`kParameterTypeDataLong` - integer value
	`kParameterTypeDataEnum` - enumerated lookup value
	`kParameterTypeDataFixed` - fixed point value
	`kParameterTypeDataDouble` - IEEE 64 bit floating point value
	`kParameterTypeDataBitField` - bit field (Boolean) value
	`kParameterTypeDataRGBValue` - RGBColor data
	`kParameterTypeDataImage` - reference to an image

This atom is required.

Parameter Alternate Data Type

This atom defines an alternative data type for the parameter. The alternative is the data type of the value stored in the parameter, if that data type is supported by the system your application is running on. If the system does not support this data type, the parameter stores a value whose data type is specified in the parameter data type atom.

For example, if the parameter alternate data type is `kParameterTypeDataColorValue`, the parameter holds a value of type `CMColor`; however, this type is only supported by systems that have the ColorSync extension loaded. On systems that do not have ColorSync, the parameter data type is used instead.

This atom's data is stored in a `ParameterAlternateDataType` data structure, which in turn relies on the `ParameterAlternateDataEntry` data structure.

```
typedef struct
{
    OSType      dataType;       // The type of the data
    QTAtomType  alternateAtom;  // The atom to use for alternate data
} ParameterAlternateDataEntry;

typedef struct
{
    long                            alternateCount;
    ParameterAlternateDataEntry     alternates[];
} ParameterAlternateDataType;
```

`dataType` This field in the `ParameterAlternateDataEntry` structure can take one of the following values:

`kParameterTypeDataColorValue` - CM color data

`kParameterTypeDataCubic` - cubic Beziers

`kParameterTypeDataNURB` - nurbs

The parameter alternate data type atom is optional.

Parameter Data Range

The Parameter Data Range atom defines the legal range of values that the parameter can take. It also defines a scaling constant that defines how the legal range of values can be translated into a range that is more suitable for display in a user interface.

The atom's data is structured as a `RangeRecord`, defined below. The exact format of this data depends on the data type of the parameter being described.

```
// 'text'
typedef struct
{
    long    maxChars;   // Maximum length of the string
    long    maxLines;   // Number of editing lines (typically 1)
} StringRangeRecord;

// 'long'
typedef struct
{
    long            minValue;           // Minimum value the long can be
    long            maxValue;           // Maximum value the long can be
    long            scaleValue;         // Scaling constant
    long            precisionDigits;    // number of digits of precision
                                        // when editing via typing
} LongRangeRecord;

// 'enum'
typedef struct
{
    long            enumID;         // The ID of the 'enum' atom in the
                                    // root container to search
} EnumRangeRecord;

// 'fixd'
typedef struct
{
    Fixed           minValue;           // Minimum value the Fixed can be
    Fixed           maxValue;           // Maximum value the Fixed can be
    Fixed           scaleValue;         // Scaling constant
    long            precisionDigits;    // number of digits of precision
                                        // when editing via typing
} FixedRangeRecord;

// 'doub'
typedef struct
{
    QTFloatDouble   minValue;           // Minimum value of parameter
    QTFloatDouble   maxValue;           // Maximum value of parameter
    QTFloatDouble   scaleValue;         // Scaling constant
    long            precisionDigits;    // number of digits of precision
                                        // when editing via typing
} DoubleRangeRecord;

// 'bool'
typedef struct
{
    long            maskValue;  // value to mask on/off to set/clear the
                                // boolean
} BooleanRangeRecord;
```

```
// 'rgb '
typedef struct
{
    RGBColor    minColor;       // Minimum value the RGBColor can be
    RGBColor    maxColor;       // Maximum value the RGBColor can be
} RGBRangeRecord;

// The RangeRecord data structure is the union of all of the above
typedef struct
{
    union
    {
        LongRangeRecord         longRange;
        EnumRangeRecord         enumRange;
        FixedRangeRecord        fixedRange;
        DoubleRangeRecord       doubleRange;
        StringRangeRecord       stringRange;
        BooleanRangeRecord      booleanRange;
        RGBRangeRecord          rgbRange;
    } u;
} RangeRecord;
```

The `minValue` and `maxValue` fields of the `DoubleRangeRecord` data structure can take, in addition to an actual `QTFloatDouble` value, the following predefined values:

- `kNoMinimumDouble` - ignore the minimum value
- `kNoMaximumDouble` - ignore the maximum value
- `kNoScaleDouble` - don’t perform any scaling of value

The `minValue` and `MaxValue` fields of the `LongRangeRecord` data structure can take, in addition to an actual long integer value, the following predefined values:

- `kNoMinimumLongFixed` - ignore minimum value
- `kNoMaximumLongFixed` - ignore maximum value
- `kNoScaleLongFixed` - don’t perform any scaling of value
- `kNoPrecision` - allow as many digits as format

The Parameter Data Range atom is required, except for group descriptions.

Parameter Data Behavior

The Parameter Data Behavior atom contains user interface hints that suggest to the client application how a parameter should be displayed.

Note
These user interface hints can be ignored by your application if you have a specific interface style you wish to implement. However, Apple recommends that you use the editing mechanisms suggested for the parameter whenever possible. If your application does not use the suggested behavior, you will present an inconsistent and potentially confusing interface to your users. ◆

```
typedef struct
{
    QTAtomID    groupID;
    long        controlValue;
} ControlBehaviors;

typedef struct
{
    OSType  behaviorType;
    long    behaviorFlags;
    union
    {
        ControlBehaviorscontrols;
    } u;
} ParameterDataBehavior;
```

`behaviorType` This field contains a value that specifies a user interface for editing the parameter's value. This field should contain one of the following pre-defined values:

`kParameterItemEditText` - the parameter should be edited using an edit text field.

`kParameterItemEditLong` - the parameter should be edited using an edit text field that only accepts numerical entries.

`kParameterItemEditFixed` - the parameter should be edited using an edit text field that accepts floating-point numerical entries.

`kParameterItemPopUp` - the parameter should be edited using a pop-up menu. This data behavior should only be used with parameters whose data type is `kParameterTypeDataEnum`; the pop-up menu is populated from the enumeration values.

`kParameterItemRadioCluster` - the parameter should be edited using a group of radio buttons. This data behavior should only be used with parameters whose data type is `kParameterTypeDataEnum`; the radio buttons are created from the enumeration values

`kParameterItemCheckBox` - the parameter should be edited using a checkbox This data behavior should only be used with parameters whose data type is `kParameterTypeDataBitField`.

`kParameterItemControl` - the parameter should be edited using a standard control appropriate to the data type of the parameter. For parameters that accept a scalar value, such as a `Fixed` or a `Long`, the control used is a slider.

`kParameterItemLine` - a horizontal line is drawn in above the control that manipulates this parameter's value.

`kParameterItemRectangle` - a rectangle is drawn around the control that manipulates this parameter's value.

`kParameterItemColorPicker` - the parameter should be edited using a color swatch and picker.

`kParameterItemGroupDivider` - start of a new group of items.

`kParameterItemStaticText` - the parameter's name is displayed as a static text field.

`kParameterItemDragImage` - the parameter should be edited as an image that accepts drag and drop entry of new images.

`kParameterItemDragPath`- the parameter should be edited as a path display that allows the user to drag out a new path.

`behaviorFlags` This field can take one or more of the following values:

`kGraphicsNoFlags` - no options for graphics.

`kGraphicsFlagsGray` - any lines or rectangles that are drawn have a grayscale appearance. If this option is not set, lines and rectangles are drawn in black.

`kGroupNoFlags` - no options for the group.

`kGroupAlignText` - the controls in the group are aligned.

`kGroupSurroundBox` - the controls in the group are surrounded with a box.

`kGroupMatrix` - display the controls in the group in a matrix, if such an arrangement is possible.

`kGroupNoName` - do not display the name of the group.

The following `behaviorFlags` values allow you to optionally show or hide a group depending on the value entered into a parameter. This allows you to express simple conditionals within a standard parameters dialog box. For example, you may want a pop-up menu with a set of fixed options, and an 'Others...' option; if the user chooses 'others', a text edit field is enabled to allows the user to enter their own value.

To do this, you can use the `kDisableWhenLessThan` flag to specify that the group containing the text control is disabled when the user chooses any value in the pop-up menu that is less than the last, 'Others...' option.

The following flags are available to control selective disabling of groups. For each of these flags, the ID of the group to be disabled is stored in the `groupID` field of the `controls` data structure. The value that is used in the comparison operation is stored in the `controlValue` field of the `controls` data structure.

`kDisableWhenNotEqual` - When the value chosen for this parameter is not equal to `controlValue`, disable the group `groupID`.

`kDisableWhenEqual` - When the value chosen for this parameter is equal to `controlValue`, disable the group `groupID`.

`kDisableWhenLessThan` - When the value chosen for this parameter is less than the `controlValue`, disable the group `groupID`.

`kDisableWhenGreaterThan` - When the value chosen for this parameter is greater than the `controlValue`, disable the group `groupID`.

Note
You can only disable groups, not individual parameters. However, you can create a group with no visual attributes that contains a single parameter. ◆

The parameter data behavior atom is required.

Parameter Data Usage

The parameter data usage atom defines the intended use of the data in the parameter. This information can be used by your application to provide a more appropriate user interface for a parameter or group of parameters. For example, if your application knows that a set of four long integer values actually represent a rectangle, it can present a graphical display of the rectangle, rather than simply displaying four numeric input fields.

The data in this atom is stored in the following data structure:

```
typedef struct
{
    OSType  usageType;
} ParameterDataUsage;
```

`usageType` This field defines the actual use that a parameter or group of parameters. It can take one of the following values:

`kParameterUsagePixels` - The parameters in the group contain a set of pixels.

`kParameterUsageRectangle` - The parameters in the group contain the top-left and bottom-right co-ordinates of a rectangle.

`kParameterUsagePoint` - The parameters in the group contain the co-ordinates of a point.

`kParameterUsage3DPoint` - The parameters in the group contain the X-Y-Z co-ordinates of a 3D point.

`kParameterUsage3by3Matrix` - The parameters in the group contain a 3x3 matrix of values.

`kParameterUsageDegrees` - The parameter contains degrees.

`kParameterUsageRadians` - The parameter contains radians.

`kParameterUsagePercent` - The parameter contains a percentage.

`kParameterUsageSeconds` - The parameter contains seconds.

`kParameterUsageMilliseconds` - The parameter contains milliseconds.

`kParameterUsageMicroseconds` - The parameter contains microseconds.

The parameter data usage atom is optional.

Parameter Data Default Item

The parameter data default item atom contains the default value for the parameter. This value is stored in a `QTAtom` and can be copied directly into the parameter; your application does not need to understand the contents or format of the atom in order to do this.

The parameter data default item atom is required, except for group descriptions.

Special Description Types

If the parameter atom type and ID atom of a parameter description contains the constant `kNoAtom`, this indicates that the value being described is not a parameter to the effect but is a group. Besides groups, two further special cases are covered in the following sections— enumeration lists and source counts.

Groups

It can sometimes be valuable to group a set of parameters together. For example, you might want to align the labels of the parameters in a group, or enclose a set of parameters with a box in the user interface. The grouping mechanism allows you to specify a set of parameters and the attributes that are applied to the group.

If the parameter data type and ID atom of a description contains child atoms, rather than data, it defines a group. A group is a set of related atoms, where the relationship amongst them can be based on attributes such as:

- layout—for example, the group is a set of text labels that should be aligned.
- spatial—for example, the items in the group should be placed side by side to optimize dialog box layout.
- naming—the items in the group are related controls that should be displayed under a single heading in the dialog box.
- usage—a pair of long integers may together specify a co-ordinate. In this case, they can be grouped together and the group's parameter data usage atom set to `kParameterUsagePoint`.

Groups can be nested within one another as needed. Groups can optionally have a name, which allows your application to place grouped parameters within a panel or tabbed group under that name.

Listing 22-27 shows an example of a group, which in this case contains a single parameter description.

Listing 22-27 An example group atom from an 'atms' resource definition.

```
kParameterAtomTypeAndID, 100, noChildren,
{
    OSType { "none" };      // Use 'none' as this is not a real parameter
    long { "0" };
    kAtomNoFlags;
    string { "" };
};

kParameterDataBehavior, 100, noChildren,
{
    kParameterItemGroupDivider; // Use a divider to separate this group
    kGroupNoFlags;
};

kParameterDataType, 100, 1*5,   // 1 parameter * 5 atoms to describe each
                                //parameter
{
};
```

```
kParameterAtomTypeAndID, 3, noChildren,
{
    OSType { "pMul" };
    long { "1" };
    kAtomNotInterpolated;
    string { "Pre-multiply color" };
};

kParameterDataType, 3, noChildren,
{
    kParameterTypeDataRGBValue;
};

kParameterDataRange, 3, noChildren,
{
    short { "0" };
    short { "0" };
    short { "0" };
    short { "65535" };
    short { "65535" };
    short { "65535" };
};

kParameterDataBehavior, 3, noChildren,
{
    kParameterItemColorPicker;
    long { "0" };
};

kParameterDataDefaultItem, 3, noChildren,
{
    short { "65535" };
    short { "65535" };
    short { "65535" };
};
```

Enumeration Lists

When an enumerated type is required for a parameter value, a new enumeration list is placed directly into the root atom container. Enumeration lists are arrays of name-and-value pairings in the following format:

```
typedef struct
{
    long    value;
    Str255  name;
} EnumValuePair;

typedef struct
{
    long            enumCount;  // number of enumeration items to follow
    EnumValuePair   values[1];  // values and names for them
} EnumListRecord;
```

The type of an enumeration list atom is `kParameterEnumList` (`'enum'`). Listing 22-28 shows an enumeration list that contains three elements.

Listing 22-28 An example enumeration list from an 'atms' resource definition.

```
kParameterEnumList, 1, noChildren,
{
    long { "3" };           // No of elements in the enum

    long {"1"}; string { "Straight Alpha" };
    long {"2"}; string { "Pre-multiply Alpha" };
    long {"3"}; string { "Reverse Alpha" };
};
```

Source Count

The source count atom (`kSourceMaxAtomType`, `'srcs'`) contains a single long integer value that defines the maximum number of sources that this effect can accept. The atom is always placed in the root atom container of the parameter description.

The source count atom is required.

Built-in QuickTime Video Effects

QuickTime 3 includes a set of built-in video effects. There are two classes of effects provided for your use:

- The SMTPE effects, which are implementations of the 133 standard effects defined by the Society of Motion Picture & Television Engineers
- A set of effects implemented by Apple Computer, which you can use for a variety of purposes.

The SMPTE Video Effects

The 133 SMPTE effects are available in four separate effect components, divided by the type of effect they implement.

SMPTE Wipe Effects ('smpt')

This effect is an implementation of the 34 wipes from ANSI / SMPTE 258M-1993, plus two Apple-defined wipes that choose a random effect. These are a series of masking or "reveal" type wipes that take place between two sources. For full definitions of these 34 wipes and what they look like, refer to the SMPTE documentation.

The SMPTE effects take two sources and seven parameters:

Percentage

Wipe ID

Soft border

Border width

Border color

Horizontal repeat

Vertical repeat

Parameters

Name	Code	Type	Description
Percentage	`'pcnt'`	`Fixed`	The frame of the effect which should be rendered, expressed as a percentage of the entire effect execution. This parameter is always tweened, which animates the effect.
Wipe ID	`'wpID'`	`Enum`	The SMPTE ID for the effect. By setting this parameter, you control which of the 47 available wipes is used.
Soft border	`'soft'`	`BitField`	If this parameter contains `true`, the border drawn around the second source is blurred.
Border width	`'widt'`	`Fixed`	The width, in pixels, of a border that is drawn around the second source.
Border color	`'bclr'`	`RGBValue`	The RGB color of the border around the second source.
Horizontal repeat	`'hori'`	`Long`	The number of horizontal repeats of the effect that are executed in a single source.
Vertical repeat	`'vert'`	`Long`	The number of vertical repeats of the effect that are executed in a single source.

Wipe ID Enum

The Wipe ID parameter can take the following values:

1. Slide horizontal

2. Slide vertical

3. Top left

4. Top right

5. Bottom right

6. Bottom left

7. Four corner

8. Four box

21. Barn vertical

22. Barn horizontal

23. Top center

24. Right center

25. Bottom center

26. Left center

41. Diagonal left down

42. Diagonal right down

43. Vertical bow tie

44. Horizontal bow tie

45. Diagonal left out

46. Diagonal right out

47.Diagonal cross

48.Diagonal box

61.Filled V

62.Filled V right

63.Filled V bottom

64.Filled V left

65.Hollow V

66.Hollow V right

67.Hollow V bottom

68.Hollow V left

71.Vertical zig zag

72.Horizontal zig zag

73.Vertical barn zig zag

74.Horizontal barn zig zag

409. Random effect - one of the 133 SMPTE effects is chosen at random.

501. Random wipe - one of the 34 SMPTE wipe effects is chosen at random.

SMPTE Iris Effects ('smp2')

This effect is an implementation of the 26 iris effects from ANSI/SMPTE 258M-1993, plus two Apple-defined wipes that choose a random effect. These are a series of “reveal” type effects that take place between two sources. For full definitions of these 26 iris effects and what they look like, please refer to the SMPTE documentation.

The SMPTE effects take two sources and seven parameters:

Percentage

Wipe ID

Soft border

Border width

Border color

Horizontal repeat

Vertical repeat

Parameters

Name	Code	Type	Description
Percentage	`'pcnt'`	`Fixed`	The frame of the effect which should be rendered, expressed as a percentage of the entire effect execution. This parameter is always tweened, which animates the effect.
Wipe ID	`'wpID'`	`Enum`	The SMPTE ID for the effect. By setting this parameter, you control which of the 26 available iris effects is used.
Soft border	`'soft'`	`BitField`	If this parameter contains `true`, the border drawn around the second source is blurred.
Border width	`'widt'`	`Fixed`	The width, in pixels, of a border that is drawn around the second source.
Border color	`'bclr'`	`RGBValue`	The RGB color of the border around the second source.
Horizontal repeat	`'hori'`	`Long`	The number of horizontal repeats of the effect that are executed in a single source.
Vertical repeat	`'vert'`	`Long`	The number of vertical repeats of the effect that are executed in a single source.

Wipe ID Enum

The Wipe ID parameter can take the following values:

101.Rectangle

102.Diamond

103.Triangle

104.Triangle right

105.Triangle upside down

106.Triangle left

107.Arrowhead

108.Arrowhead right

109.Arrowhead upside down

110.Arrowhead left

111.Pentagon

112.Pentagon upside down

113.Hexagon

114.Hexagon side

119.Circle

120.Oval

121.Oval side

122.Cat eye

123.Cat eye side

124.Round rect

125.Round rect side

127.4 point star

128.5 point star

129.6 point star

130.Heart

131.Keyhole

409. Random effect - one of the 133 SMPTE effects is chosen at random.

502. Random iris - one of the 26 SMPTE iris effects is chosen at random.

SMPTE Radial Effects ('smp3')

This effect is an implementation of the 39 radial effects from ANSI/SMPTE 258M-1993, plus two Apple-defined wipes that choose a random effect. These are a series of radial "reveal" type effects that take place between two sources. For full definitions of these 39 radial effects and what they look like, please refer to the SMPTE documentation.

The SMPTE effects take two sources and seven parameters:

Percentage

Wipe ID

Soft border

Border width

Border color

Horizontal repeat

Vertical repeat

Parameters

Name	Code	Type	Description
Percentage	`'pcnt'`	`Fixed`	The frame of the effect which should be rendered, expressed as a percentage of the entire effect execution. This parameter is always tweened, which animates the effect.
Wipe ID	`'wpID'`	`Enum`	Contains the SMPTE ID for the effect. By setting this parameter, you control which of the 39 available radial effects is used.
Soft border	`'soft'`	`BitField`	If this parameter contains `true`, the border drawn around the second source is blurred.

Name	Code	Type	Description
Border width	`'widt'`	`Fixed`	The width, in pixels, of a border that is drawn around the second source.
Border color	`'bclr'`	`RGBValue`	The RGB color of the border around the second source.
Horizontal repeat	`'hori'`	`Long`	The number of horizontal repeats of the effect that are executed in a single source.
Vertical repeat	`'vert'`	`Long`	The number of vertical repeats of the effect that are executed in a single source.

Wipe ID Enum

The Wipe ID parameter can take the following values:

201.Rotating top

202.Rotating right

203.Rotating bottom

204.Rotating left

205.Rotating top bottom

206.Rotating left right

207.Rotating quadrant

211.Top to bottom 180°

212.Right to left 180°

213.Top to bottom 90°

214.Right to left 90°

221.Top 180°

222.Right 180°

223.Bottom 180°

224.Left 180°

225.Counter rotating top bottom

226.Counter rotating left right

227.Double rotating top bottom

228.Double rotating left right

231.V Open top

232.V Open right

233.V Open bottom

234.V Open left

235.V Open top bottom

236.V Open left right

241.Rotating top left

242.Rotating bottom left

243.Rotating bottom right

244.Rotating top right

245.Rotating top left bottom right

246.Rotating bottom left top right

251.Rotating top left right

252.Rotating left top bottom

253.Rotating bottom left right

254.Rotating right top bottom

261.Rotating double center right

262.Rotating double center top

263.Rotating double center top bottom

264.Rotating double center left right

409. Random effect - one of the 133 SMPTE effects is chosen at random.

503. Random radial - one of the 39 SMPTE radial effects is chosen at random.

SMPTE Matrix Effects ('smp4')

This effect is an implementation of the 34 matrix effects from ANSI/SMPTE 258M-1993, plus two Apple-defined wipes that choose a random effect. These are a series of matrix "reveal" type effects that take place between two sources. For full definitions of these 34 matrix effects and what they look like, please refer to the SMPTE documentation.

The SMPTE effects take two sources and seven parameters:

Percentage

Wipe ID

Soft border

Border width

Border color

Horizontal repeat

Vertical repeat

Parameters

Name	Code	Type	Description
Percentage	`'pcnt'`	`Fixed`	The frame of the effect which should be rendered, expressed as a percentage of the entire effect execution. This parameter is always tweened, which animates the effect.
Wipe ID	`'wpID'`	`Enum`	Contains the SMPTE ID for the effect. By setting this parameter, you control which of the 34 available matrix effects is used.
Soft border	`'soft'`	`BitField`	If this parameter contains `true`, the border drawn around the second source is blurred.

Name	Code	Type	Description
Border width	`'widt'`	`Fixed`	The width, in pixels, of a border that is drawn around the second source.
Border color	`'bclr'`	`RGBValue`	The RGB color of the border around the second source.
Horizontal repeat	`'hori'`	`Long`	The number of horizontal repeats of the effect that are executed in a single source.
Vertical repeat	`'vert'`	`Long`	The number of vertical repeats of the effect that are executed in a single source.

Wipe ID Enum

The Wipe ID parameter can take the following values:

301.Horizontal matrix

302.Vertical matrix

303.Top left diagonal matrix

304.Top right diagonal matrix

305.Bottom right diagonal matrix

306.Bottom left diagonal matrix

310.Clockwise top left matrix

311.Clockwise top right matrix

312.Clockwise bottom right matrix

313.Clockwise bottom left matrix

314.Counter clockwise top left matrix

315.Counter clockwise top right matrix

316.Counter clockwise bottom right matrix

317.Counter clockwise bottom left matrix

320.Vertical start top matrix

321.Vertical start bottom matrix

322.Vertical start top opposite matrix

323.Vertical start bottom opposite matrix

324.Horizontal start left matrix

325.Horizontal start right matrix

326.Horizontal start left opposite matrix

327.Horizontal start right opposite matrix

328.Double diagonal top right matrix

329.Double diagonal bottom right matrix

340.Double spiral Top matrix

341.Double spiral bottom matrix

342.Double spiral left matrix

343.Double spiral right matrix

344.Quad spiral vertical matrix

345.Quad spiral horizontal matrix

350.Vertical waterfall left matrix

351.Vertical waterfall right matrix

352.Horizontal waterfall left matrix

353.Horizontal waterfall right matrix

409. Random effect - one of the 133 SMPTE effects is chosen at random.

504. Random matrix - one of the 34 SMPTE matrix effects is chosen at random.

Video Effects from Apple

The following video effects are supplied by Apple Computer and are built into QuickTime 3.

Alpha Compositor ('blnd')

This effect is used to combine two images using the alpha channels of the images to control the blending. It provides for the standard alpha blending options, and can handle pre-multiplying by any color, although white and black are most common and often run faster.

The alpha compositor effect takes a maximum of two sources and has two parameters:

Blend mode

Pre-multiply color

Parameters

Name	Code	Type	Description
Blend mode	`'bMod'`	`Enum`	Contains the blend mode for the effect.
Pre-multiply color	`'mclr'`	`RGBValue`	If the blend mode is "pre-multiply alpha," this parameter contains the color used in the pre-multiply blend, otherwise it is ignored.

Blend Mode Enum

The blend mode parameter can contain one of the following values:

1. **Straight alpha -** perform a standard alpha blend. The alpha channel value of the first source defines the amount of the first source that is included in the composited image, and one minus the alpha channel value of the first source defines the amount of the second source that is included in the composited image.
2. **Pre-multiply alpha -** calculates the destination pixel according to the following formulae:

```
DestinationRed = PreMultiplyRed * (1-alphaC) + temp1 * alphaC
DestinationGreen = PreMultiplyGreen * (1-alphaC) + temp2 * alphaC
DestinationBlue = PreMultiplyBlue * (1-alphaC) + temp3 * alphaC
```

where:

```
    alphaC = alphaB + (1-alphaB) * alphaA
    temp1 = (alphaA * SourceARed + alphaB * sourceBRed)/alphaC
    temp2 = (alphaA * SourceAGreen + alphaB * sourceBGreen)/alphaC
    temp3 = (alphaA * SourceABlue + alphaB * sourceBBlue)/alphaC
```

3. **Reverse alpha -** perform a reverse alpha blend. The one minus the alpha channel value of the first source defines the amount of the first source that is included in the composited image, and the alpha channel value of the first source defines the amount of the second source that is included in the composited image.

Alpha Gain filter ('gain')

The alpha gain filter is used to alter the alpha channel of a single source. This operation is commonly applied before passing the source to the alpha compositor effect described above. The following equation describes the alteration that is made to the source’s alpha channel:

```
newAlpha = bottomPin <= (gain*oldAlpha + offset) <= topPin
```

This means that to increase the alpha channel by a set amount, you set the gain parameter to 1.0, and the offset to the desired increase. Similarly, to increase the alpha channel by a fixed percentage, set the offset to 0.0 and the gain to the percentage increase desired. The `topPin` and `bottomPin` parameters allow you to set upper and lower bounds on the value of the alpha channel, respectively.

The alpha gain filter effect takes a maximum of one source and has four parameters:

Gain value

Offset value

Top alpha pin

Bottom alpha pin

Parameters

Name	Code	Type	Description
Gain value	`'gain'`	`Fixed`	This value is multiplied by the original alpha channel value.
Offset value	`'offs'`	`Fixed`	This value is added to the old alpha channel, after it has been multiplied by the gain parameter.
Top alpha pin	`'pinT'`	`Fixed`	The maximum value that the alpha channel can take after the gain and offset parameters have been applied.
Bottom alpha pin	`'pinB'`	`Fixed`	The minimum value that the alpha channel can take after the gain and offset parameters have been applied.

Blur Filter ('blur')

This effect applies a convolution blur effect to a single source. The actual blur that is applied is determined by the convolution kernel. This is a matrix of values that are applied to each pixels of the source to produce the destination.

The Blur effect takes a maximum of one source and has two parameters:

Amount of blurring

Brightness

Parameters

Name	Code	Type	Description
Amount of blurring	`'ksiz'`	`enum`	The size of the blur kernel to apply. This value must be one of 3, 5, 7, 9, 11, 13 or 15. The larger the kernel, the longer the effect will take to run and the greater the degree of blurring.
Brightness	`'ksum'`	`Fixed`	This is the total value of the elements of the blur kernel. Normally this value will be 1.0, which blurs the source but doesn't change its brightness. If the value is between 0.0 and 1.0, the brightness is decreased, if the value is greater than 1.0, the brightness is increased.

Chroma Key ('ckey')

The chroma key effect combines two sources by replacing all the pixels of the first source that are the specified color with the corresponding pixels of the second source. This allows the second source to "show through" the first in those places where the first source is the given color.

The chroma key effect takes a maximum of two sources and has one parameter, key color.

Parameter

Name	Code	Type	Description
Key color	`'keyc'`	`RGBValue`	The chroma key color to replace in the first source with pixels from the second source.

Cloud ('clou')

The cloud effect uses a fractal noise generator to simulate a cloud formation. This can be transparently overlaid on an image. The cloud formation’s colors can be controlled, and the cloud randomly changes shape over time.

The cloud effect takes no sources and has two parameters:

Cloud color

Background color

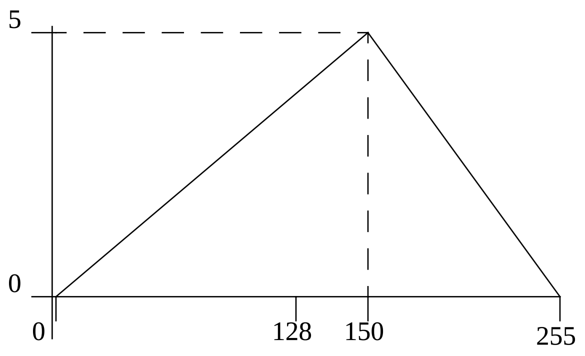
Parameters

Name	Code	Type	Description
Cloud color	`'fgdc'`	`RGBValue`	The foreground color of the cloud.
Background color	`'bckc'`	`RGBValue`	The background color of the cloud.

Color Style ('solr')

The color style effect allows you to apply two color stylizations to a single source. They are:

- Solarization - adjusts the color balance of the source by constructing a table of replacement color values from two parameters. These parameters are the maximum color intensity and the peak point of the color spread. The table starts at zero intensity and increases to the maximum intensity at the peak point. After that it falls back to zero.For example, if the color values range from 0 to 255, the maximum intensity is 5 and the peak point is at 150, the resulting table’s profile will look like:



- Posterization - reduces the number of colors in an image by replacing all pixels whose color is in a consecutive range with the middle color from that range. This produces a “color banding” effect.

Both these effects work on a per-channel basis, which means that the red, green and blue components of each pixels are independently passed through the respective algorithm.

For solarization, a maximum intensity of 1 and a peak point of 128 are the most commonly used values.

The color style effect takes a maximum of one source and has three parameters:

Solarize amount

Solarize point

Posterize amount

Parameters

Name	Code	Type	Description
Solarize amount	`'solr'`	`Long`	The maximum intensity of the solarization table.
Solarize point	`'solp'`	`Long`	The peak point of the solarization table.
Posterize amount	`'post'`	`Long`	The number of colors that are grouped and replaced with the mid-range color.

ColorSync filter ('sync')

The color sync filter adjusts the color balance of an image to match a specified color sync profile. Typically, you would use this to adjust the color profile of an image to match the current display device. This allows you to maintain accurate color representations across devices. You specify both the color sync profile of the source image and the color sync profile of the destination device the image will be rendered to.

The color sync filter takes a maximum of one source and has two parameters:

Source profile

Destination profile

Parameters

Name	Code	Type	Description
Source profile	`'srcP'`	`enum`	The color sync profile of the source image.
Destination profile	'destP'	`enum`	The color sync profile of the target device.

Color Tint filter ('tint')

The color tint filter converts its source into greyscale, then applies a light and a dark color to the image. The light color replaces the white in the greyscale image, and the dark color replaces the black. This filter also includes brightness and contrast controls. The end result is a tinted duochrome version of the source image.

You can use this filter to, for example, apply a sepia tone to a source.

The color tint filter takes a maximum of one source and has four parameters:

Dark color

Light color

Brightness

Contrast

Parameters

Name	Code	Type	Description
Dark color	`'back'`	`RGBValue`	The color to use to replace the black of the greyscale image.
Light color	`'fore'`	`RGBValue`	The color to use to replace the white of the greyscale image.
Brightness	`'brig'`	`Long`	The amount to adjust the brightness of the source by, ranging from -255 (all colors are replaced with black) to 255 (all colors are replaced with white). A value of 0 will leave the brightness unchanged.
Contrast	`'cont'`	`Long`	The amount to adjust the contrast of the source by, ranging from -255 (minimum contrast) to 255 (maximum contrast). A value of 0 will leave the contrast unchanged.

Cross Fade ('dslv')

A “cross fade” or “dissolve” provides a smooth alpha blending between two video sources, which changes over time to give a smooth fade out from the first source into the second.

This effect takes a maximum of two sources, and has a single parameter, percentage.

Parameter

Name	Code	Type	Description
Percentage	`'pcnt'`	`Fixed`	The frame of the effect which should be rendered, expressed as a percentage of the entire effect execution. This parameter is always tweened, which animates the effect.

Edge Detection Filter ('edge')

This effect applies an edge detection convolution to a single source. The performance of the edge detection is determined by the convolution kernel. This is a matrix of values applied to each pixel of the source to produce the resulting image.

This effect takes a maximum of one source, and has two parameters:

Edge thickness

Colorize

Parameters

Name	Code	Type	Description
Edge thickness	`'ksiz'`	`enum`	The size of the kernel to apply. This value must be one of 3, 5, 7, 9, 11, 13 or 15. Larger kernels will produce thicker edges in the resulting image.
Colorize	`'colz'`	`BitField`	If this parameter is set to true, the edges produces are assigned color, based on the color of the source pixels around them. Otherwise, edges are rendered as light grey against a dark grey background.

Emboss Filter ('embs')

This effect applies an emboss convolution to a single source. The performance of the embossing operation is determined by the convolution kernel. This is a matrix of values applied to each pixel of the source to produce the resulting image.

This effect takes a maximum of one source, and has one parameter, amount of embossing.

Parameter

Name	Code	Type	Description
Amount of embossing	`'ksiz'`	`Fixed`	The size of the kernel to apply. This value must be one of 3, 5, 7, 9, 11, 13 or 15. Larger kernels will produce a more heavily embossed result.

Explode ('xplo')

In an explode effect, source B grows from a single point expanding out until it entirely covers source A. The centre point of the explosion is defined in the effect parameters.

This effect takes a maximum of two sources, and has three parameters:

Percentage

Explode centre X

Explode centre Y

Parameters

Name	Code	Type	Description
Percentage	`'pcnt'`	`Fixed`	The frame of the effect which should be rendered, expressed as a percentage of the entire effect execution. This parameter is always tweened, which animates the effect.
Explode centre X	`'xcnt'`	`Fixed`	The x-coordinate of the explosion centre.
Explode centre Y	`'ycnt'`	`Fixed`	The y-coordinate of the explosion centre.

Film Noise Filter ('fmns')

The film noise filter alters a single source, simulating some of the effects that are seen on aged film stock. This effect can be used to transform a video source into one that looks like it was shot on film that has suffered the effects of age and mishandling.

The specific features, which can be controlled independently, are:

- Hairs. These are a simulation of hairs lying on the surface of the film. Each hair is randomly generated, and is colored in a randomly chosen shade of light grey.
- Scratches. These are vertical or near-vertical one-pixel lines drawn onto the destination image that simulate scratches in the film. Each scratch lasts for a pre-calculated length of time. During its lifespan the scratch's position is randomly peturbed. Shortly before the scratch is removed, it will begin to shorten. The color of the scratches is a randomly chosen shade of light grey.
- Dust. These simulate dust particles on the surface of the film. Dust particles are drawn using the same algorithm that generates the hairs, but the particles are more tightly curled, and drawn in a darker shade of grey.
- Film fade. This simulates an overall change in the color of the film stock. Every pixel of the source image is passed through the film fade algorithm, so this can be processor-intensive.

The film noise effect takes a single source and has eight parameters:

Hair density

Hair length

Scratch density

Scratch duration

Scratch width

Dust density

Dust size

Film fade

Parameters

Name	Code	Type	Description
Hair density	'hden'	Long	This parameter controls the number of hairs that are drawn on each frame and the frequency with which hairs appear. The maximum number of hairs per frame is a randomly generated number between 1 and the value of this parameter. The chance of each hair appearing on a single frame is 1 in (the value of this parameter).
Hair length	'hlen'	Long	The maximum length (in pixels) of the hairs being drawn.
Scratch density	'sden'	Long	This parameter controls the number of scratches that are drawn on each frame and the frequency with which scratches appear. The maximum number of scratches per frame is a randomly generated number between 1 and the value of this parameter. The chance of each scratch appearing on a single frame is 1 in (the value of this parameter).
Scratch duration	'sdur'	Long	The maximum number of frames that each scratch appears for. The actual number of frames for each scratch is a randomly chosen value between 1 and this value plus 20.

Name	Code	Type	Description
Scratch width	`'swid'`	`Long`	The maximum width, in pixels, of a scratch. The actual width is a randomly chosen number between 1 and this value.
Dust density	`'dden'`	`Long`	This parameter controls the number of dust particles that are drawn on each frame and the frequency with which dust particles appear. The maximum number of dust particles per frame is a randomly generated number between 1 and the value of this parameter. The chance of each dust particle appearing on a single frame is 1 in (the value of this parameter).
Dust size	`'dsiz'`	`Long`	The size of each dust particle.
Film fade	`'fade'`	`Enum`	The type of film fade effect (if any) to apply.

The Film Fade Enum

The film fade parameter can take one of the following values:

1. None—the destination image is a copy of the source.
2. Sepia tone—the destination is a slightly saturated, monochromatic version of the source, colorized into shades of sepia (a light red-brown).
3. Black and white—the destination is a greyscale version of the source.
4. Faded color film—the destination is a color de-saturated version of the source.
5. 1930’s color film - the destination is a color super-saturated version of the source.

Fire ('Fire')

The fire effect simulates a fire by generating a number of individual flames of randomized appearance. You can control various parameters that define how the flames are generated and how they change over time.

The fire effect takes no sources and has four parameters:

Spread rate

Sputter rate

Water rate

Restart rate

Parameters

Name	Code	Type	Description
Spread rate	`'sprd'`	`Long`	How quickly the fire expands to its highest level from its starting point. The higher the value, the more quickly the fire starts up and reaches its maximum burn rate.
Sputter rate	`'decy'`	`Long`	How quickly the flames die down as they move up the screen. Low numbers result in very tall flames, high numbers in very low flames.
Water rate	`'watr'`	`Long`	How often “water” is tossed on the base of the fire, instantly putting out the fire at that point. High numbers result in a fire that's very broken up (i.e. many areas of burning and non-burning) while lower numbers result in a wider, smoother fire.
Restart rate	`'rset'`	`Long`	How often entire fire is put out, then allowed to restart.

General Convolution Filter ('genk')

This effect applies a general purpose convolution effect to a single source. The effect that results is completely determined by the values entered into the kernel parameters of the effect. The kernel for this convolution is always a 3-by-3 matrix of values.

The values of the cells of the convolution kernel determine the value that is assigned to each pixel of the destination frame. The convolution algorithm examines every pixel of the source, and the eight pixels surrounding it. These values are multiplied by the appropriate values in the cells and summed. This sum is then used as the value of the corresponding destination pixel.

Many computer graphics textbooks offer a more complete and rigorous explanation of convolution, and you are encouraged to consult these for useful values that the kernel can take.

As an example of a kernel that produces a useful result, the following kernel values will shift an image left by one pixel:

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

The General Convolution Filer effect takes a maximum of one sources and has nine parameters:

Cell one

Cell two

Cell three

Cell four

Cell five

Cell six

Cell seven

Cell eight

Cell nine

Parameters

Name	Code	Type	Description
Cell one	`'cel1'`	`Fixed`	The value to be placed into the first cell of the kernel.
Cell two	`'cel2'`	`Fixed`	The value to be placed into the second cell of the kernel.
Cell three	`'cel3'`	`Fixed`	The value to be placed into the third cell of the kernel.
Cell four	`'cel4'`	`Fixed`	The value to be placed into the fourth cell of the kernel.
Cell five	`'cel5'`	`Fixed`	The value to be placed into the fifth cell of the kernel.
Cell six	`'cel6'`	`Fixed`	The value to be placed into the sixth cell of the kernel.
Cell seven	`'cel7'`	`Fixed`	The value to be placed into the seventh cell of the kernel.
Cell eight	`'cel8'`	`Fixed`	The value to be placed into the eighth cell of the kernel.
Cell nine	`'cel9'`	`Fixed`	The value to be placed into the ninth cell of the kernel.

The nine cells in the kernel are laid out as follows:

Cell one	Cell two	Cell three
Cell four	Cell five	Cell six
Cell seven	Cell eight	Cell nine

Gradient Wipe ('matt')

The gradient wipe effect transitions between two sources, with the change pattern controlled by an input image. At the start of the effect, the area covered by the input image shows the first source, while the area outside the input image shows the second. Over the duration of the effect, the input image is shrunk until only the second source is visible.

The gradient wipe effect takes two sources and has two parameters:

Percentage

Matte image

Parameters

Name	Code	Type	Description
Percentage	`'pcnt'`	`Fixed`	The frame of the effect which should be rendered, expressed as a percentage of the entire effect execution. This parameter is always tweened, which animates the effect.
Matte image	`'matt'`	`Image`	The matte image that controls the transition between the two sources.

HSL Balance Filter ('hsvb')

This filter effect allows you to independently adjust the hue, saturation and lightness (also known as value or brightness) channels of a single source. The effect adjusts every pixel in the source, multiplying the hue component of the pixel by the value of the hue multiplier parameter, the saturation component by the value of the saturation multiplier parameter, and so on.

The HSL balance filter effect takes one source and has three parameters:

Hue multiplier

Saturation multiplier

Lightness multiplier

Parameters

Name	Code	Type	Description
Hue multiplier	`'hmul'`	`Fixed`	The amount to adjust the hue channel value of each pixel by.
Saturation multiplier	`'smul'`	`Fixed`	The amount to adjust the saturation channel value of each pixel by.
Lightness multiplier	`'vmul'`	`Fixed`	The amount to adjust the lightness channel value of each pixel by.

Implode ('mplo')

In an implode effect, source A shrinks down to a single point, revealing source B. The center point of the implosion is defined in the effect parameters.

The implode effect takes a maximum of two sources and has three parameters:

Percentage

Implode centre X

Implode centre Y

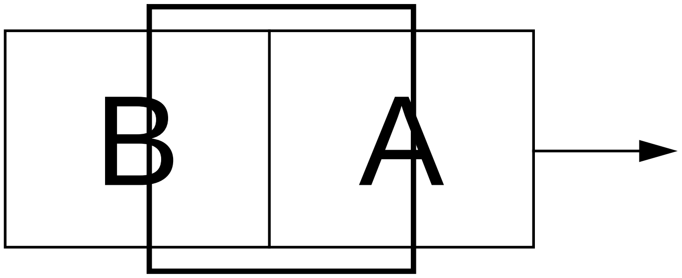
Parameters

Name	Code	Type	Description
Percentage	`'pcnt'`	`Fixed`	The frame of the effect which should be rendered, expressed as a percentage of the entire effect execution. This parameter is always tweened, which animates the effect.
Implode centre X	`'xcnt'`	`Fixed`	The x-coordinate of the implosion centre.
Implode centre Y	'ycnt'	`Fixed`	The y-coordinate of the implosion centre.

Push ('push')

A push is an effect where one source image replaces another, with both sources moving at the same time. For example, source A would typically occupy the entire screen, and then source B would slide in from the left, while source A slides out to the right at the same time. Figure 22-6 shows an example of a push effect in progress.

Figure 22-6 An example Push effect



The push effect takes a maximum of two sources and has two parameters:

Percentage

From direction

Parameters

Name	Code	Type	Description
Percentage	`'pcnt'`	`Fixed`	The frame of the effect which should be rendered, expressed as a percentage of the entire effect execution. This parameter is always tweened, which animates the effect.
From direction	`'from'`	`Enum`	Contains the direction from which source B will replace source A

From Direction Enum

The from direction parameter can contain the following values:

1. Top
2. Right
3. Bottom
4. Left

RGB Balance Filter ('rgbb')

The RGB balance filter allows you to independently adjust the alpha, red, green and blue channels of a single source. The effect adjusts every pixel in the source, multiplying the red component of the pixel by the value of the red multiplier parameter, the green component by the value of the green multiplier parameter, and so on.

The RGB balance filter takes a maximum of one source and has three parameters:

Red multiplier

Green multiplier

Blue multiplier

Parameters

Name	**Code**	**Type**	**Description**
Red multiplier	`'rmul'`	`Fixed`	The amount to adjust the red channel value of each pixel by.
Green multiplier	`'gmul'`	`Fixed`	The amount to adjust the green channel value of each pixel by.
Blue multiplier	`'bmul'`	`Fixed`	The amount to adjust the blue channel value of each pixel by.

Ripple ('ripl')

The ripple effect simulates a pool of water that overlays an image. The area within the ripple mask will undulate, giving the appearance of water. If the user clicks within the ripple area, concentric waves are sent across the water, simulating a stone dropped into the pool.

The ripple effect takes no sources and has one parameter, ripple mask.

Parameters

Name	Code	Type	Description
Ripple mask	`'mask'`	`Image`	A 1-bit image that acts as a mask; the ripple effect is seen at every pixel corresponding to a pixel in the mask that is set.

Sharpen Filter ('shrp')

This effect applies a convolution sharpen effect to a single source. The sharpening that is applied is determined by the convolution kernel. This is a matrix of values that are applied to each pixels of the source to produce the destination.

The sharpen filter effect takes one source and has two parameters:

Amount of sharpening

Brightness

Parameters

Name	Code	Type	Description
Amount of sharpening	`'ksiz'`	`enum`	The size of the sharpen kernel to apply. This value must be one of 3, 5, 7, 9, 11, 13 or 15. The smaller the kernel, the faster the effect will run and the greater the degree of sharpening.

Name	Code	Type	Description
Brightness	`'ksum'`	`Fixed`	This is the total value of the elements of the sharpen kernel. Normally this value will be 1.0, which sharpens the source but doesn't change its brightness. If the value is between 0.0 and 1.0, the brightness is decreased, if the value is greater than 1.0, the brightness is increased.

Slide ('slid')

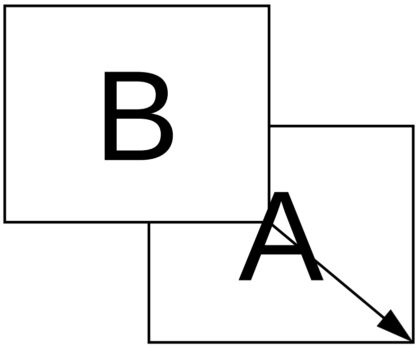
In an slide effect source B slides onto the screen to cover source A. At the end of the effect, source B will completely cover source A. The angle from which source B enters the frame is stored in a parameter, with 0 being the top of the screen. Figure 22-7 shows an example slide effect.

The slide effect takes a maximum of two sources and has two parameters:

Percentage

Slide angle

Figure 22-7 An example Slide effect



Parameters

Name	Code	Type	Description
Percentage	'pcnt'	Fixed	The frame of the effect which should be rendered, expressed as a percentage of the entire effect execution. This parameter is always tweened, which animates the effect.
Slide angle	'angl'	Fixed	The angle from which source B will enter the frame. This value is expressed in degrees, with 0 degrees defined as the top of the screen.

QuickTime Video Effects Reference

This section details the constants, data types and functions used to support video effects in QuickTime. The APIs for dealing with effects are split into two sections: high-level functions that present a simplified interface for common tasks, and low-level functions that provide complete access to the effects functionality.

The functions presented in this chapter are the effect-specific calls introduced in QuickTime 3. To make full use of video effects, you will need to use other QuickTime APIs, such as track creation and editing functions.

Constants

This section describes the constants defined in QuickTime to support video effects.

Effects List Atom Names

These constants specify the four character codes for the two atom types in the atom container returned by calls to `QTGetEffectsList`.

```
enum {
    kEffectNameAtom = FOUR_CHAR_CODE('name'),
    kEffectTypeAtom = FOUR_CHAR_CODE('type')
}
```

Constant descriptions

`kEffectNameAtom`	The code for the atom containing the name of an entry in the effects list.
`kEffectTypeAtom`	The code for the atom containing the type of an entry in the effects list.

Effect Action Selectors

These constants specify the action selectors you can pass to the functions `QTStandardParameterDialogDoAction` and `ImageCodecStandardParameterDialogDoAction`.

```
enum {
    pdActionConfirmDialog,
    pdActionSetAppleMenu,
    pdActionSetEditMenu,
    pdActionGetDialogValues,
    pdActionSetPreviewUserItem,
    pdActionSetPreviewPicture,
    pdActionSetColorPickerEventProc,
    pdActionSetDialogTitle
    pdActionGetSubPanelMenu,
    pdActionActivateSubPanel,
    pdActionConductStopAlert
}
```

Constant descriptions

`pdActionConfirmDialog`

Retrieves the current parameter values from the standard parameters dialog box. The parameter values are placed in the `parameters` parameter that was passed to the function that created the dialog box.

`pdActionSetAppleMenu`
: Passes the menu handle for the current Apple menu to the standard parameters dialog box.

`pdActionSetEditMenu`
: Passes the menu handle for your application's Edit menu to the standard parameters dialog box.

`pdActionGetDialogValues`
: Retrieves the current parameter values from the standard parameters dialog box. This parameter values are placed in the `params` parameter of the `QTStandardParameterDialogDoAction` or `ImageCodecStandardParameterDialogDoAction` function.

`pdActionSetPreviewUserItem`
: Passes the number of the user item resource that should be replaced by the effect preview movie clip.

`pdActionSetPreviewPicture`
: Sets the images that are used in the preview window of the standard parameters dialog box.

`pdActionSetColorPickerEventProc`
: Sets the function that will be used when the standard parameters dialog box needs to display a color picker control.

`pdActionSetDialogTitle`
: Sets the title of the standard parameters dialog box.

`pdActionGetSubPanelMenu`
: Returns a `menuHandle` containing the pop-up menu used to switch between multiple parameter panels. You can parse this data structure to derive a list of the panels that are being used by the standard parameters dialog box.

`pdActionActivateSubPanel`
: Sets the currently active panel of the standard parameter dialog box.

`pdActionConductStopAlert`
: Displays a Stop Alert containing a message.

Get Options for QTGetEffectsList

These constants define the flags that can be passed in the `getOptions` parameter of the `QTGetEffectsList` function.

```
enum {
    elOptionsIncludeNoneInList
}
```

Constant descriptions

`elOptionsIncludeNoneInList`
: Includes the “none” effect in the list of effects returned by `QTGetEffectsList`.

Standard Parameter Dialog Box Options

These constants are used to control how parameter values are entered into a standard parameters dialog box that is generated when you call the function `ImageCodecCreateStandardParameterDialog`.

```
enum {
    pdOptionsCollectOneValue,
    pdOptionsAllowOptionalInterpolations
}
```

Constant descriptions

`pdOptionsCollectOneValue`
: This value indicates that only one value can be entered for parameters that can be tweened.

`pdOptionsAllowOptionalInterpolations`
: This value indicates that parameters that are flagged as `kAtomInterpolateIsOptional` in their parameter description will be displayed with a user interface that allows the entry of tweened values.

ImageCodecValidateParameters Options

These are the values that can be passed in the `validationFlags` parameter of a call to `ImageCodecValidateParameters`.

```
enum {
    kParameterValidationNoFlags,
    kParameterValidationFinalValidation
}
```

Constant descriptions

`kParameterValidationNoFlags`	This value indicates that a standard validation should take place. `ImageCodecValidateParameters` is being called because the user has changed the value of a parameter in the standard parameters dialog box.
`kParameterValidationFinalValidation`	This value indicates that this validation is the final validation before the standard parameters dialog box is dismissed. This is useful if you want to perform a single validation of the parameters just after the user clicks the OK button to dismiss the dialog box.

Real Time Flag

This is the value that an effect should return when `QTGetEffectSpeed` is called and the component can run the effect in real time.

```
enum {
    effectIsRealtime
}
```

Constant descriptions

`effectIsRealtime`	The effect can process frames in real time.

Data Types

This section describes the data types that are relevant to calling and creating effects.

Parameter Dialog Box Preview Image Specifier

This data structure contains a picture that will be used as one of the preview images in the preview window of the standard parameters dialog box. The preview window shows previews of the effects the user chooses in the dialog box. QuickTime provides a default images, but you can replace one or more of these by calling `QTStandardParameterDialogDoAction` (or its low-level equivalent, `ImageCodecStandardParameterDialogDoAction`) with a `pdActionSetPreviewPicture` action selector.

The `sourceID` numbers correspond to the sources an effect uses. For example, an effect that uses one source will use the preview image with `sourceID` set to 1, while a two source effect will use preview pictures 1 and 2.

```
struct QTParamPreviewRecord {
    long        sourceID;
    PicHandle   sourcePicture;
};
typedef struct QTParamPreviewRecord QTParamPreviewRecord;
typedef QTParamPreviewRecord *QTParamPreviewPtr;
```

Field descriptions

`sourceID`	The number of the preview image.
`sourcePicture`	The preview image itself.

Effect Source Descriptors

These data structures describe the sources to an effect. The `SourceData` data structure contains a pointer to either raw image compression manager image data or to an effect that acts as the source. The `EffectSourcePtr` data structure holds information about the type of source, and pointers to the track data of the effect and to the next source in the input chain.

```
typedef struct EffectSource     EffectSource;
typedef EffectSource            *EffectSourcePtr;

union SourceData {
    CDSequenceDataSourcePtr image;
    EffectSourcePtr     effect;
};
typedef union SourceData SourceData;
```

Field descriptions

`image`	A pointer to the raw source image data.
`effect`	A pointer to the effect.

```
struct EffectSource {
    long                effectType;
    Ptr                 data;
    SourceData          source;
    EffectSourcePtr     next;
};
```

Field descriptions

`effectType`	The type of the effect or the constant `kEffectRawSource` if the source is raw image compression manager data.
`data`	A pointer to the track data for the effect.
`source`	The source itself.
`next`	A pointer to the next source in the input chain.

Effect Frame Description

This data structure contains the parameters of a single frame of an effect. It is passed to the `MyEffectBegin` and `MyEffectRenderFrame` functions to describe the frame to be rendered.

```
struct EffectsFrameParams {
    ICMFrameTimeRecord  frameTime;
    long                effectDuration;
    Boolean             doAsync;
    unsigned char       pad[3];
    EffectSourcePtr     source;
    void *              refCon;
};
typedef struct EffectsFrameParams EffectsFrameParams;
typedef EffectsFrameParams *EffectsFrameParamsPtr;
```

Field descriptions

frameTime	Timing data for the current frame. This record includes information such as the total number of frames being rendered in this sequence, and the current frame number.

`effectDuration`	The duration of a single effect frame.
`doAsync`	This field contains `true` if the effect can process asynchronously.
`source`	A pointer to the input sources.
`refCon`	A pointer to storage for this instantiation of the effect.

The Decompression Parameters Structure

Several fields of the decompression parameters structure (`CodecDecompressParams`) are relevant to effects, and in particular are used when you are writing your own effect component. This section describes only those fields of this data structure that apply to effects; for a complete description of the decompression parameters structure, see Chapter 4, “Image Compressor Components.”

```
struct CodecDecompressParams {
    …
    CodecCapabilities           *capabilities;
    …
    ICMFrameTimePtr             frameTime;
    …
    UInt16                      majorSourceChangeSeed;
    UInt16                      minorSourceChangeSeed;
    CDSequenceDataSourcePtr     sourceData;
    …
    OSType                      **wantedDestinationPixelTypes;
    …
    Boolean                     needUpdateOnTimeChange;
    …
    Boolean                     needUpdateOnSourceChange;
    …
    long                        requestedBufferWidth;
    long                        requestedBufferHeight;
};
typedef struct CodecDecompressParams CodecDecompressParams;
```

Field descriptions

`capabilities`	A pointer to a `CodecCapabilities` data structure containing the required capabilities of the effect.

`frameTime`	The current frame time. The information in this parameter can be used to calculate the current frame number, relative to the total duration of the effect.
`majorSourceChangeSeed`	An integer value that is incremented each time a data source is added or removed. This provides a fast way for an effect component to know when its data sources have changed.
`minorSourceChangeSeed`	An integer value that is incremented each time a data source is added or removed, or the data contained in any of the sources changes.This provides a fast way for an effect component to know when a source changes.
`sourceData`	A pointer to a `CDSequenceDataSource` structure that contains the head of a linked list of all data sources. Because each data source contains a link to the next data source, an effect component can access all data sources from this field.
`wantedDestinationPixelTypes`	A handle to a zero-terminated list of non-RGB pixel formats that the component can decompress. Set this value to `nil` if your component does not support non-RGB pixel spaces. If your effect component supports non-RGB pixel formats, your component's `MyEffectSetup` function should place the list of them into this field.The Image Compression Manager (ICM) copies this data structure, so it is up to your component to dispose of it later. It is suggested that components allocate this handle during the `Open` routine and dispose of it during the `Close` routine.
`needUpdateOnTimeChange`	Setting this field to `true` indicates to the ICM that your component's `EffectBegin` and `EffectRenderFrame` functions should be called whenever the time of the movie changes.The default value of this field is `true`.
`needUpdateOnSourceChange`	Setting this field to `true` indicates to the ICM that your component's `EffectBegin` and `EffectRenderFrame` functions should be called whenever one or more of the sources to the effect changes. The default value of this field is `false`.

`requestedBufferWidth`
: Specifies the width of the image buffer to use, in pixels. For this value to be used, the `codecWantsSpecialScaling` flag in the `CodecCapabilities` record must be set.

`requestedBufferHeight`
: Specifies the height of the image buffer to use, in pixels. For this value to be used, the `codecWantsSpecialScaling` flag in the `CodecCapabilities` record must be set.

High-Level Functions

These functions give you high-level pre-packaged access to the video effects architecture. These functions are designed to be easy to use and give you access to the most common uses of the QuickTime Video Effects architecture.

QTGetEffectsList

The `QTGetEffectsList` function returns a `QTAtomContainer` holding a list of the currently installed effects components. You can constrain the type of effect that is added to the list by this function.

```
OSErr QTGetEffectsList(
                    QTAtomContainer *returnedList,
                    long minSources,
                    long maxSources,
                    QTEffectListOptions getOptions);
```

`returnedList`
: If `QTGetEffectsList` returns `noErr`, this parameter contains a newly created `QTAtomContainer` holding a list of their currently installed effects. Any data stored in the parameter on entry is overwritten by the list of effects. It is the responsibility of the calling application to dispose of the storage by calling `QTDisposeAtomContainer` once the list is no longer required.

`minSources`
: The minimum number of sources that an effect must have to be added to the list. Pass `-1` as this parameter to specify no minimum.

`maxSources`	The maximum number of sources that an effect can have to be added to the list. Pass `-1` as this parameter to specify no maximum.
`getOptions`	Options that control which effects are added to the list. The values that `getOptions` can take are: `0` - Include every effect, except the "none" effect (and any prohibited by the values of `minSources` and `maxSources`) `elOptionsIncludeNoneInList` - Include the "none" effect in the returned list. The "none" effect does nothing.

DESCRIPTION

This function returns the list of currently installed effects in the `returnedList` parameter. The returned list contains two atoms for each effect component. The first atom, of type `kEffectNameAtom` contains the name of the effect. The second atom, of type `kEffectTypeAtom` contains the type of the effect, which is the sub-type of the effect component. This list is sorted alphabetically on the names of the effects.

The `minSources` and `maxSources` parameters allow you to restrict which effects are returned in the list, by specifying the minimum and maximum number of sources that qualifying effects can have. You can also specify whether the "none" effect is included in the list using the `getOptions` parameter.

SPECIAL CONSIDERATIONS

The calling application is responsible for disposing of the `QTAtomContainer` returned in `returnedList`. The application should do this by calling `QTDisposeAtomContainer` once it has finished using the parameters description.

RESULT CODES

`ParamErr`	–50	The value passed to `returnedList` was not a valid `QTAtomContainer`.
Any memory error	n/a	Memory errors returned by `QTNewAtomContainer` can also be returned by `QTGetEffectsList`.

QTCreateStandardParameterDialog

The `QTCreateStandardParameterDialog` function creates a dialog box that allows the user to choose an effect from the list of effects passed to the function. The user can also set the values of the parameters of the effect chosen. The standard dialog includes a preview of the effect.

```
OSErr QTCreateStandardParameterDialog(QTAtomContainer effectList,
                    QTAtomContainer parameters,
                    QTParameterDialogOptions dialogOptions,
                    QTParameterDialog *createdDialog);
```

`effectList`	A list of the effects that the user can choose from. In most cases you should call `QTGetEffectsList` to generate this list. If you pass `NIL` in this parameter, `QTCreateStandardParameterDialog` calls `QTGetEffectsList` to retrieve the list of all currently installed effects; this list is then presented to the user.
`parameters`	An effect description containing the default parameter values for the effect. If the effect named in the parameter description is in `effectlist`, that effect is displayed when the dialog is first shown, and its parameter values are set from the parameter description.
`dialogOptions`	Options to control the behavior of the dialog. The valid options are the same as those used by `ImageCodecCreateStandardParameterDialog` (see "ImageCodeCreateStandardParameterDialog" beginning on page 734 for details).
`createdDialog`	Returns a reference to the dialog box that is created by this function. You pass this value to the high-level functions `QTIsStandardParameterDialogEvent` and `QTDismissStandardParameterDialog`.

DESCRIPTION

This function creates and displays a standard parameter dialog box that allows the user to choose an effect from the list in the `effectsList` parameter. The dialog box also allows the user to choose values for the parameters of the effect, to preview the effects as they choose and customize them, and to get more information about each effect.

Your application must call the `WaitNextEvent` and `QTIsStandardParameterDialogEvent` functions to allow the user to interact with the dialog box that is shown.

Note
The dialog box will remain hidden until the first event is processed by `QTIsStandardParameterDialogEvent`. At this point, the dialog box will be displayed. ◆

The list of effects passed in the `effectList` parameter should be in the format defined in “QTGetEffectsList” (page 725). You should use the `QTGetEffectsList` function to generate this list.

The `parameters` parameter returns an effect description data structure, as described in “Creating an Effect Description” (page 612). If you pass an empty `QTAtomContainer` as this parameter, the dialog box will be showing the first effect in `effectsList` when it appears, otherwise it will display the effect named in the effect description.

The value returned in the `createdDialog` parameter is a reference to the dialog box that is created. You should only pass this value to the `QTIsStandardParameterDialogEvent` and `QTDismissStandardParameterDialog` functions.

You can modify the default behavior of the dialog box that is created by calling `QTStandardParameterDialogDoAction` (see “QTStandardParameterDialogDoAction” (page 730).

RESULT CODES

`ParamErr`	–50	The `createdDialog` parameter contained an illegal value. This parameter must contain the address of a `QTParameterDialog`.
`NoCodecErr`	–8961	The `effectsList` parameter does not contain a valid list of effects.
Any memory error	n/a	Memory errors returned by `QTNewAtomContainer` can also be returned by `QTGetEffectsList`.

QTIsStandardParameterDialogEvent

The `QTIsStandardParameterDialogEvent` function determines if an event returned from `WaitNextEvent` is processed by a standard parameter dialog box created by `QTCreateStandardParameterDialog`.

```
OSErr QTIsStandardParameterDialogEvent(
                    EventRecord *pEvent,
                    QTParameterDialog createdDialog);
```

`pEvent`	The event returned from a call to `WaitNextEvent`.
`createdDialog`	The reference to the standard parameters dialog box that is returned by `QTCreateStandardParameterDialog`.

DESCRIPTION

This function determines whether an operating system event, as returned by a call to the `WaitNextEvent` function, is processed by the standard parameters dialog box created by `QTCreateStandardParameterDialog`. If the event is processed by the dialog box it is handled by this function, otherwise it is an event that your application is responsible for processing.

After you create a standard parameter dialog box, you must pass every event received from `WaitNextEvent` through the `QTIsStandardParameterDialogEvent` function to determine if your application should handle the event. Once the dialog box has been confirmed or cancelled by the user, you should no longer call the `QTIsStandardParameterDialogEvent` function.

You determine the status of the event by examining the return code of the `QTIsStandardParameterDialogEvent` function. This can be one of the following values:

`featureUnsupported`	The event was not handled by the standard parameter dialog box, so your application should handle it in the usual way.
`noErr`	The event was handled by the standard parameter dialog box, so you should not process it yourself; instead you should call `WaitNextEvent` to retrieve further events.

`codecParameterDialogConfirm`
: The event was handled by the standard parameter dialog box, and the user has clicked the OK button of the dialog box. The effect description that you passed in the parameters parameter to the QTIsStandardParameterDialogEvent function has been filled to reflect the choices the user made in the dialog box. You should call the QTDismissStandardParameterDialog function to close the dialog box.

`userCanceledErr`
: The event was handled by the standard parameter dialog box, and the user has clicked the Cancel button of the dialog box. You should call the `QTDismissStandardParameterDialog` function to close the dialog box.

QTDismissStandardParameterDialog

The `QTDismissStandardParameterDialog` function closes a standard parameter dialog box that was created using the `QTCreateStandardParameterDialog` function. All memory associated with the dialog box is disposed of.

```
OSErr QTDismissStandardParameterDialog(QTParameterDialog createdDialog);
```

`createdDialog` The reference to the standard parameters dialog box that is returned by `QTCreateStandardParameterDialog`.

QTStandardParameterDialogDoAction

The `QTStandardParameterDialogDoAction` function allows you to change some of the default behaviors of the standard parameter dialog box.

```
OSErr QTStandardParameterDialogDoAction(
                     QTParameterDialog createdDialog,
                     long action,
                     void *params);
```

`createdDialog`	The reference to the dialog box created by calling `QTCreateStandardParameterDialog`.
`action`	Determines which of the actions supported by this function will be performed.
`params`	The (optional) parameter to the action. The type passed in this parameter depends on the value of the `action` parameter.

DESCRIPTION

The `QTStandardParameterDialogDoAction` function allows you to change some of the default behaviors of a standard parameter dialog box you create using the `QTCreateStandardParameterDialog` function. To choose which of the available customizations to perform, pass an **action selector** value in the `action` parameter and, optionally, a single parameter in `params`.

The following list details the valid action selectors:

`pdActionSetAppleMenu`
: Use this selector to make the Apple menu available while the standard parameter dialog box is active. Pass the `MenuHandle` of the Apple menu in the `params` parameter.

`pdActionSetEditMenu`
: Use this action to make your application's Edit menu available while the standard parameters dialog box is active.Pass the `MenuHandle` of the Edit menu in the `params` parameter.

`pdActionSetPreviewPicture`
: Use this action to change the images that are used in the preview window of the standard parameters dialog box. QuickTime provides default images but you may wish, for example, to use thumbnail images taken from your application instead. The `params` parameter contains a `QTParamPreviewPtr` referencing the image to use (see "Parameter Dialog Box Preview Image Specifier" (page 721)).

`pdActionSetDialogTitle`
: Sets the title of the standard parameters dialog box. The `params` parameters contains a Pascal `Str255` string.

`pdActionGetSubPanelMenu`
: Returns a list of the sub-panels used by the standard parameters dialog box. If there are more parameter controls than can fit into

the available area of the dialog box, the parameters are automatically split into sub-panels (usually by segmenting the set of parameters by group). The pop-up menu used to switch between the panels is returned as a `MenuHandle` in the `params` parameter.

`pdActionActivateSubPanel`.
Sets the current sub-panel. The `params` parameter contains a long integer that is the number of the panel to be displayed.

`pdActionConductStopAlert`.
Displays a Stop alert. The `params` parameter contains a `StringPtr` that is displayed in the alert. This feature can be used in conjunction with the `ImageCodecValidateParameters` function to inform the user when invalid parameter values have been entered.

QTGetEffectSpeed

The `QTGetEffectSpeed` function returns the speed of the effect, expressed in frames-per-second.

```
OSErr QTGetEffectSpeed(
                QTAtomContainer parameters,
                Fixed *pFPS);
```

`parameters`	Contains parameter values for the effect.
`pFPS`	The speed of the effect is returned in this parameter. Speed is expressed in frames per second.

DESCRIPTION

This function returns the execution speed of an effect in frames-per-second. The value returned should not be treated as an absolute measurement of effect performance. In particular, most effects only return one value, regardless of parameter settings and hardware. This value is an estimate of execution speed on a reference hardware platform. Actual performance will vary depending on hardware, configuration and parameter options.

Effects can also return the pre-defined constant `effectIsRealtime` as their speed. This means that the effect will execute as quickly as frames can be sent to it for rendering.

Low-level Functions

These functions provide more complex and comprehensive interfaces to the effects dialog functionality. Using the low-level functions, you can gain more control over the standard parameters dialog box, for example allowing you to incorporate user interface elements from the dialog box into your own application-defined dialog box.

ImageCodecGetParameterList

The `ImageCodecGetParameterList` function returns a parameter description data structure for the specified component instance.

```
ComponentResult ImageCodecGetParameterList(
                          ComponentInstance ci,
                          QTAtomContainer *parameterDescription);
```

`ci`	An instance of an effect component.
`parameterDescription`	The returned parameters description data structure for this component instance.

DESCRIPTION

This function returns the parameter description for the component instance `ci`. The parameter description is a set of `QTAtom` data structures stored inside a `QTAtomContainer`. Each parameter of the effect is described in the parameter description, with details of its name, type, legal values and hints about how a user interface to the parameter should be constructed.

For full details of the set of atoms stored in a parameter description, see “The Parameter Description Format” (page 668).

SPECIAL CONSIDERATIONS

The calling application is responsible for disposing of the `QTAtomContainer` returned in `parameterDescription`. The application should do this by calling `QTDisposeAtomContainer` once it has finished using the parameter description.

ImageCodeCreateStandardParameterDialog

This function creates a parameters dialog box for the specified effect. The dialog box allows the user to set the parameter values for the effect. You can optionally request that the controls from the dialog box are included within a dialog box of the calling application.

```
ComponentResult ImageCodecCreateStandardParameterDialog (
                    ComponentInstance ci,
                    QTAtomContainer parameterDescription,
                    QTAtomContainer parameters,
                    QTParameterDialogOptions  dialogOptions,
                    DialogPtr existingDialog,
                    short existingUserItem,
                    QTParameterDialog *createdDialog);
```

`ci`	An instance of an effect component. The dialog box that is created will allow the user to specify the parameters of this effect.
`parameterDescription`	The parameter description for this effect. You can obtain a valid `parameterDescription` by calling `ImageCodecGetParameterList`.
`parameters`	The container that will receive the user's chosen parameter values once the dialog has been dismissed.
`dialogOptions`	Control over how parameters containing tween data are presented in the created dialog box. If `dialogOptions` contains `0`, two values are collected for each parameter that can be tweened, and the usual tweening operation will be performed for the duration of the effect being controlled.

If `dialogOptions` contains `pdOptionsCollectOneValue`, only one value can be entered for any parameters that can be tweened. This means that parameter values will not be tweened when the effect is played back.

If `dialogOptions` contains `pdOptionsAllowOptionalInterpolations`, parameters that are defined as `kAtomInterpolateIsOptional` in their parameter description will be allowed to take tweened values. This allows you to force the “advanced” user interface mode described in “Parameter Atom Type and ID” (page 668).

`existingDialog`

An existing dialog box that will have the controls from the standard parameters dialog box added to it. This parameter may contain `nil` if you want `ImageCodecCreateStandardParameterDialog` to create a stand-alone dialog box. If `existingDialog` is not `nil`, the controls that would make up the dialog box are instead added to the existing dialog box.

`existingUserItem`

The number of the user item in the existing dialog box that should be replaced with controls from the standard parameter dialog box. You should only pass a value to this parameter if the `existingDialog` parameter is not `nil`.

`createdDialog` On exit, this parameter contains a reference to the dialog created and displayed by the function. This reference is required by several other low-level effects functions. It will contain a valid dialog identifier even if you requested that the controls from the standard parameter dialog box are incorporated into an existing dialog box.

DESCRIPTION

This function can to create a stand-alone standard parameter dialog box, or it can add the controls from the standard parameter dialog box to an existing dialog box provided by the calling application.

ImageCodecIsStandardParameterDialogEvent

This function processes events related to a standard parameters dialog box.

```
ComponentResult ImageCodecIsStandardParameterDialogEvent(
                    ComponentInstance ci,
                    EventRecord *pEvent,
                    QTParameterDialog createdDialog);
```

`ci`	An instance of an effect component. This must be the instance that was passed to `ImageCodecCreateStandardParameterDialog` to create the dialog box.
`pEvent`	A pointer to an event record returned by `WaitNextEvent`.
`createdDialog`	A reference to the dialog box created by the call to `ImageCodecCreateStandardParameterDialog`.

DESCRIPTION

After you call `ImageCodecCreateStandardParameterDialog` to create a standard parameter dialog box, you must pass every non-null event return from `WaitNextEvent` to `ImageCodecIsStandardParameterDialogEvent`. The function processes events related to the standard parameter dialog box, passing other events to your application for processing.

`ImageCodecIsStandardParameterDialogEvent` returns an error code that indicates whether the event pointed to by `pEvent` was processed or not. If the error code returned is `featureUnsupported`, your application should process the event in the normal way. If the `ImageCodecIsStandardParameterDialogEvent` returns any other value, the event was processed or an error occurred.

RESULT CODES

`featureUnsupported`	The event was not processed by the function. Your application should process the event normally.
`codecParameterDialogConfirm`	The user clicked the OK button in the dialog box. Your application should call `ImageCodecStandardParameterDialogDoAction`, then `ImageCodecDismissStandardParameterDialog`.
`userCanceledErr`	The user clicked the Cancel button in the dialog box. Your application should call `ImageCodecDismissStandardParameterDialog`.

ImageCodecStandardParameterDialogDoAction

This function allows you to retrieve values from a standard parameter dialog box. You should only call this function if you incorporated the user interface elements of the standard parameter dialog box into your own dialog box. If you created a stand-alone standard parameters dialog box, this function is automatically called for you when the user clicks the OK button.

```
ComponentResult ImageCodecStandardParameterDialogDoAction(
                        ComponentInstance ci,
                        QTParameterDialog createdDialog,
                        long action,
                        void *params);
```

`ci`	An instance of an effect component. This must be the same instance as was passed to `ImageCodecCreateStandardParameterDialog` to create the dialog box.
`createdDialog`	A reference to the dialog box created by the call to `ImageCodecCreateStandardParameterDialog`.
`action`	The action selector, which determines which of the available actions you want the function to perform.
`params`	The (optional) parameter to the action. The type passed in this parameter depends on the value of the `action` parameter.

DESCRIPTION

The `ImageCodecStandardParameterDialogDoAction` function allows you to change the default behavior of the standard parameter dialog box, and provides a mechanism for your application to communicate with controls that were incorporated into an application dialog box. It also allows you to retrieve parameter values from the dialog box at any time.

You specify which function will be performed by passing an action selector in the `action` parameter and, optionally, a single parameter in `params`.

The following list describes the available action selectors:

`pdActionSetDialogTitle`
: Sets the title of the standard parameters dialog box from the `params` parameter, which contains a Pascal `Str255` string.

`pdActionConfirmDialog`
: Retrieves the current parameter values from the standard parameters dialog box. The parameter values are placed in the `parameters` parameter that was passed to the appropriate call to `ImageCodecCreateStandardParameterDialog`.

`pdActionSetAppleMenu`
: Use this selector to make the Apple menu available while the standard parameter dialog box is active. Pass the `MenuHandle` of the Apple menu in the `params` parameter.

`pdActionSetEditMenu`
: Use this action to make your application's Edit menu available while the standard parameters dialog box is active.Pass the `MenuHandle` of the Edit menu in the `params` parameter.

`pdActionGetDialogValues`
: Retrieves the current parameter values from the standard parameters dialog box. The parameter values are placed in the `QTAtomContainer` you pass in the `params` parameter to this call.

`pdActionSetPreviewUserItem`
: Passes the number of the user item that should be replaced by the effect preview movie clip. The `params` parameter contains a long integer holding the user item number.

`pdActionSetPreviewPicture`
: Use this action to change the images that are used in the preview window of the standard parameters dialog box.

QuickTime provides default images but you may wish, for example, to use thumbnail images taken from your application instead. The `params` parameter contains a `QTParamPreviewPtr` referencing the image to use (see "Parameter Dialog Box Preview Image Specifier" (page 721)).

`pdActionSetColorPickerEventProc`
Set the function that will be used when the standard parameters dialog box needs to display a color picker. The `params` parameter should contain a `UserEventUPP` that points to the replacement color picker function.

`pdActionSetDialogTitle`
Sets the title of the standard parameters dialog box. The `params` parameters should contain a Pascal `Str255` string.

`pdActionGetSubPanelMenu`
Returns a list of the sub-panels used by the standard parameters dialog box. If there are more parameter controls than can fit into the available area of the dialog box, the parameters are automatically split into sub-panels (usually by segmenting the set of parameters by group). The pop-up menu used to switch between the panels is returned as a `MenuHandle` in the `params` parameter.

`pdActionActivateSubPanel`
Sets the current sub-panel. The `params` parameter contains a long integer that is the number of the panel to be displayed.

`pdActionConductStopAlert`
Displays a Stop alert. The `params` parameter contains a `StringPtr` that is displayed in the alert. This feature can be used in conjunction with the `ImageCodecValidateParameters` function to inform the user when invalid parameter values have been entered.

ImageCodecDismissStandardParameterDialog

This function retrieves values from a standard parameter dialog box, then closes the dialog box.

```
ComponentResult ImageCodecDismissStandardParameterDialog(
                         ComponentInstance ci,
                         QTParameterDialog createdDialog);
```

`ci`	An instance of an effect component. This must be the same instance as was passed to `ImageCodecCreateStandardParameterDialog` to create the dialog box.
`createdDialog`	A reference to the dialog box created by the call to `ImageCodecCreateStandardParameterDialog`.

DESCRIPTION

This function should be called after the `ImageCodecIsStandardParameterDialogEvent` function returns `codecParameterDialogConfirm` or `userCanceledErr`, which indicate that the user has dismissed the dialog box.

The function dismisses the dialog box, deallocating any memory allocated during the call to the `ImageCodecCreateStandardParameterDialog` function.

Component-Defined Functions

This section defines the effect-specific functions that you may supply in your effect components. This section is only of interest to developers who are creating their own effects components; if you are writing an application that uses QuickTime Video Effects, you can skip this section.

The functions defined in this section are those called by the Component Manager through your component’s dispatch function (see “What Effects Components Do” (page 639)).

Note
The interfaces to these functions are described assuming you are using Apple's component dispatch helper code and sample effect Framework as described in "The Sample Effect Component" (page 661). Apple strongly recommends that you re-use these code samples where possible when implementing your own effect components. ◆

If you are using the sample effect component, you can use the default implementations of several of these functions in most circumstances.

MyEffectSetup

The Component Manager calls this function when a sequence of frames is about to be rendered.

```
ComponentResult MyEffectSetup(
                    EffectGlobals *glob,
                    CodecDecompressParams *decompressParams);
```

`glob`	A pointer to the effect's global data structure.
`decompressParams`	Information about the sequence that is about to be decompressed.

DESCRIPTION

This function is called immediately before a client application (which may be QuickTime) is going to call your component to render a sequence of frames of your effect.

Your component should examine the `capabilities` field of the `decompressParams` data structure to ensure that it can meet the requirements for executing this sequence. In particular, it should check the bit depth and pixel format requirements of the sequence. If the sequence requires a bit depth and pixel format combination that your component does not support, this function should return the nearest supported combination in the `decompressParams->capabilities` field. In this case, QuickTime will redirect all source and destination bitmaps through offscreen graphics worlds that have the bit depth and pixel format characteristics that you specify.

MyEffectBegin

The Component Manager calls this function to request that your component prepare to render a single frame of its effect.

```
ComponentResult MyEffectBegin(
                     EffectGlobals *glob,
                     CodecDecompressParams *decompressParams,
                     EffectsFrameParamsPtr effect);
```

`glob`	A pointer to the effect's global data structure.
`decompressParams`	Information about the current sequence of frames.
`effect`	The parameters describing this frame.

DESCRIPTION

This function is called by a client application (which may be QuickTime) immediately before your `MyEffectRenderFrame` function. Your `MyEffectBegin` function should ensure that the information it holds about the current source and destination buffers and the parameter values for the effect are valid. If any of these have changed since the last call to `MyEffectBegin`, the new values should be read from the appropriate data structures.

This function is guaranteed to be called synchronously. In particular, this means you can allocate and move memory, and call functions that allocate or move memory.

MyEffectRenderFrame

The Component Manager calls this function to request that your component render a single frame of its effect.

```
ComponentResult MyEffectRenderFrame(
                    EffectGlobals *glob,
                    EffectsFrameParamsPtr effect);
```

`glob`	A pointer to the effect's global data structure.
`effect`	The parameters describing this frame.

DESCRIPTION

This function is called by a client application (which may be QuickTime) when your effect component needs to render a single frame of your effect. This function contains the implementation of your effect.

▲ **WARNING**
This function is *not* guaranteed to be called synchronously. In particular, this means your function implementation must not allocate or move memory, or call any function that allocates or moves memory. ▲

MyEffectCancel

The Component Manager calls this function to stop processing of the current effect.

```
ComponentResult MyEffectCancel(
                    EffectGlobals *glob,
                    EffectsFrameParamsPtr effect);
```

`glob`	A pointer to the effect's global data structure.
`effect`	The parameters describing this frame.

DESCRIPTION

This function is called by a client application (which may be QuickTime) to halt the rendering of the current sequence of frames before the last frame has been rendered. If your component is running synchronously, it should simply return `noErr`; no further calls to your `MyEffectRenderFrame` function will be made for this sequence.

If your component is running asynchronously, this function should dequeue all outstanding render frame requests, then return `noErr`.

MyEffectGetCodecInfo

The Component Manager calls this function to request information about the component.

```
ComponentResult MyEffectGetCodecInfo(
                        EffectGlobals *glob,
                        CodecInfo *info);
```

`glob`	A pointer to the effect's global data structure.
`info`	A pointer to the data structure that will contain the codec information.

DESCRIPTION

This function is called by a client application (which may be QuickTime) to request information about your effect component. Your function should fill out the `CodecInfo` data structure passed to it. You can use the `GetComponentResource` function to retrieve a `'cdci'` resource that stores this information, if you have provided one in your component.

RESULT CODES

`noErr`	0	The function successfully filled out the `info` field.
`paramErr`	–50	Your function should return this value if the `info` parameter contains `nil`.

MyEffectGetParameterListHandle

The Component Manager calls this function to request a parameter description for this component.

```
ComponentResult MyEffectGetParameterListHandle(
                        EffectGlobals *glob,
                        Handle theHandle);
```

`glob`	A pointer to the effect's global data structure.
`theHandle`	A pointer to a handle that will contain the parameter description of this effect.

DESCRIPTION

This function is called by a client application (which may be QuickTime) to request a parameter description for your effect. This function can use the `GetComponentResource` function to retrieve an `'atms'` resource that stores this information, if you have provided one in your component.

MyEffectGetSpeed

The Component Manager calls this function to request information about the rendering speed of this effect component.

```
long MyEffectGetSpeed(
                        EffectGlobals *glob,
                        QTAtomContainer parameters,
                        Fixed *pFPS)
```

`glob`	A pointer to the effect's global data structure.
`parameters`	The current parameter values for this effect.
`pFPS`	A pointer to a `Fixed` value that will contain the rendering speed of this effect on exit.

DESCRIPTION

This function is called by a client application (which may be QuickTime) to request information about the rendering speed of your effect. This function should return a `Fixed` value in `pFPS`, which represents the rendering speed in frame per second of the effect.

If your effect can render in real time, it should return a value of `effectIsRealtime`. Otherwise, you should return an estimate of the number of frames your effect can render per second. Because rendering speeds are hardware-dependent, effect authors can choose to measure actual rendering speeds in this function. Alternatively, effect authors can choose to return a single value for all hardware configurations, estimating the value for a reference hardware platform.

Apple recommends that the values returned are rounded down to the nearest common frames-per-second value, such as 15, 24 or 30.

MyEffectValidateParameters

If your effect implements this optional function, the Component Manager calls it whenever the user changes a parameter value in the standard parameter dialog box, or attempts to dismiss the dialog.

```
ComponentResult MyEffectValidateParameters(
                    EffectGlobals *glob
                    QTAtomContainer parameters,
                    QTParameterValidationOptions  validationFlags,
                    StringPtr errorString);
```

`glob`	A pointer to the effect’s global data structure.
`parameters`	The current parameter values for this effect.
`validationFlags`	Flags that indicate whether a parameter value has changed or the user is dismissing the standard parameter dialog box.
`errorString`	A `StringPtr` that is contains an error string explaining to the user why the validation has failed.

DESCRIPTION

This optional function is called by a client application (which may be QuickTime) when your effect's standard parameter dialog box is being displayed. It can be called in two circumstances: if the user changes a parameter value in the dialog box; if the user dismisses the dialog box.

The purpose of this function is to allow your effect to validate its parameters. The current parameter values are passed to the effect in `parameters`. If all of these values are valid, this function should return `noErr`. Otherwise, you should return a `paramErr` and put an explanatory message in the `errorString` parameter.

Data Codec Components

This chapter describes data codec components, which are new to QuickTime 3. **Data codec components** enable you to compress and decompress data using a specified compressor component. For information about codecs, see *Inside Macintosh: QuickTime Components*.

About Data Codec Components

Data compressor components have a component type of `DataCompressorComponentType` and data decompressor components have a component type of `DataDecompressorComponentType`. Each component has a unique component subtype.

With the `DataCodecCompress` function, you can compress data using a specified compressor component. The `DataCodecDecompress` function lets you decompresses data using a specified compressor component.

You can also compress and decompress sprites and 3D media samples by calling the `DataCodecCompress` and `DataCodecDecompress` functions.

Your software calls the `DataCodecBeginInterruptSafe` function before performing a compression or decompression operation during interrupt time. When your software has finished a compression or decompression operation that must be performed during interrupt time, it can release any memory of other resources used by the `DataCodecBeginInterruptSafe` function to make the operation safe by calling the `DataCodecEndInterruptSafe` function.

Using Data Codec Components

To open a connection to a data compressor or decompressor component, you use the `OpenADefaultComponent` function. To iterate through all available data compressors or decompressors, you use the `FindNextComponent` function.

To decompress data using a data decompressor component, you use the `DataCodecDecompress` routine.

Compressing or Decompressing During Interrupt Time

If a compressor or decompressor component implements the `DataCodecBeginInterruptSafe` and `DataCodecEndInterruptSafe` functions, your application or other software can perform compression or decompression operations during interrupt time. You do this as follows:

1. Before performing the compression or decompression operation, call the `DataCodecBeginInterruptSafe` function.In the call, pass the maximum size of a data block to be compressed or decompressed in the `maxSrcSize` parameter.
2. If the call fails, do not perform operations that are not safe during interrupt time.
3. When the compression or decompression operation is complete, call the `DataCodecEndInterruptSafe` function to release resources used to make the operation safe at interrupt time.

Data Codec Components Reference

Functions

DataCodecCompress

The `DataCodecCompress` function compresses data using the specified compressor component.

```
pascal ComponentResult DataCodecCompress (
                        DataCodecComponent dc,
                        void *srcData,
                        UInt32 srcSize,
                        void *dstData,
                        UInt32 dstBufferSize,
                        UInt32 *actualDstSize,
                        UInt32 *decompressSlop );
```

`dc`	Specifies the compressor component to used.
`srcData`	Contains a pointer to the data to be compressed.
`srcSize`	Contains the size of the data to be compressed, in bytes.
`dstData`	Contains a pointer to the buffer in which to store the compressed data.
`dstBufferSize`	Contains the size of the buffer in which to store the compressed data, in bytes.
`actualDstSize`	Contains the size of the compressed data that was created, in bytes.

`decompressSlop`
Contains the number of bytes that should be added to the decompression buffer size if decompression occurs in place.

DISCUSSION

Before calling the `DataCodecCompress` function, you should call the `DataCodecGetCompressBufferSize` function to obtain the maximum possible size of the compressed data that will be returned. You can then use this value as the value of the `dstBufferSize` parameter. Note that a buffer for compressed data that is the same size as the uncompressed data may not be large enough: in some cases, the size of the compressed data can be larger than the size of the decompressed data.

When you compress data, you should store the size of data before compression at the beginning of the file, immediately before the compressed data. This allows you to obtain the size of the decompressed data and allocate the buffer for storing the decompressed data before calling the `DataCodecDecompress` function.

You can compress sprites by calling the `DataCodecCompress` function. If you do, you must

- Include the uncompressed size of the sample at the beginning of the sample, before the compressed data.
- Store the component subtype of the data compressor used to compress the sprite in the `decompressorType` field of the sample's sprite sample description.

You can compress 3D media samples by calling the `DataCodecCompress` function. If you do, you must

- Include the uncompressed size of the sample at the beginning of the sample, before the compressed data.
- Store the component subtype of the data compressor used to compress the sprite in the `decompressorType` field of the sample's sprite sample description.

DataCodecDecompress

The `DataCodecDecompress` function decompresses data using the specified compressor component.

```
pascal ComponentResult DataCodecDecompress (
                     DataCodecComponent dc,
                     void *srcData,
                     UInt32 srcSize,
                     void *dstData,
                     UInt32 dstBufferSize );
```

`dc`	Specifies the decompressor component to used.
`srcData`	Contains a pointer to the data to be decompressed.
`srcSize`	Contains the size of the data to be decompressed, in bytes.
`dstData`	Contains a pointer to the buffer in which to store the decompressed data.
`dstBufferSize`	Contains the size of the buffer in which to store the decompressed data, in bytes.

DISCUSSION

Before allocating the buffer in which to store decompressed data, you need to get the size of the decompressed data. The size is normally stored at the beginning of the file, immediately before the compressed data.

Performing Compression and Decompression at Interrupt Time

DataCodecBeginInterruptSafe

Your software calls the `DataCodecBeginInterruptSafe` function before performing a compression or decompression operation during interrupt time.

```
pascal ComponentResult  DataCodecBeginInterruptSafe (
                        DataCodecComponent dc,
                        unsigned long maxSrcSize);
```

`dc`	The instance of a compressor or decompressor component for this request. Your software obtains this reference when calling the Component Manager's `OpenComponent` or `OpenDefaultComponent` function.
`maxSrcSize`	The maximum size of a block of data to be compressed or decompressed, in bytes.

DISCUSSION

If the function returns an error, your software must not perform compression or decompression operations during interrupt time.

This function allocates any temporary buffers that are necessary to perform the operation during interrupt time. To release the resources used to make the operation safe during interrupt time, call the `DataCodecEndInterruptSafe` function or close the instance of the compressor or decompressor component.

DataCodecEndInterruptSafe

When your software has finished a compression or decompression operation that must be performed during interrupt time, it can release any memory of other resources used by the `DataCodecBeginInterruptSafe` function to make the operation safe by calling the `DataCodecEndInterruptSafe` function.

```
pascal  ComponentResult DataCodecEndInterruptSafe (
                        DataCodecComponent dc);
```

`dc`	The instance of a compressor or decompressor component for this request.

QuickTime Vectors

This chapter discusses QuickTime vectors. **QuickTime vectors** are mathematical descriptions of images that are smaller in size than their equivalent bitmaps and can be scaled without loss of image quality. A vector QT atom container specifies the characteristics of a QuickTime vector. The container contains atoms for paths and can also contain atoms that specify attributes of the paths, such as their color.

This chapter presents you with general information about QuickTime vectors. It also provides background information about Macintosh QuickDraw GX objects that you need to know before using QuickTime vectors in your software.

Read this chapter if you need to create and manipulate QuickTime vectors. This chapter shows you, for example, how to convert QuickDraw GX data to QuickTime vectors by using the QuickDraw GX vector transcoder. It also provides you with a “QuickTime Vector Reference” (page 870) that describes the constants, data types, and functions associated with QuickTime vectors.

About QuickTime Vectors

QuickTime vectors are mathematical descriptions of images. Images described using QuickTime vectors can be scaled without loss of quality. In addition, the description of these images can be much smaller than equivalent images represented with bitmaps, even when the bitmaps are compressed.

Data for a QuickTime vector is stored in a QT atom container. The contents are a stream of atoms. Each atom contains one of the following:

- a specification of a path
- a style specification for the path, such as the stroke width of the pen used to draw the path

- an atom that identifies the end of the data stream

In the current implementation, each path can include up to 32767 contours, and each contour can include up to 32767 points that define it.

An atom that contains a style specification determines the style of any paths in the stream that follow it unless or until it is replaced by another atom of the same type. If there is no atom for a particular style at any point the data stream, a default value is used for that style. The default value is replaced if and when an atom for the style appears in the data stream.

QuickTime Vectors and QuickDraw GX

Many of the data structures for QuickTime vectors are derived from data structures for QuickDraw GX, the advanced graphics architecture developed by Apple Computer, Inc. Because of this, many of the identifiers include the prefix "gx". However, you do not need QuickDraw GX to use QuickTime vectors; all its functionality is included in QuickTime. You do need QuickDraw GX or the QuickDraw GX extension to convert existing QuickDraw GX data to QuickTime vector data with the QuickDraw GX transcoder component, as described in "Converting QuickDraw GX Data to QuickTime Vectors" (page 869).

QuickDraw GX Concepts and Structures

To use QuickTime vectors, you need to understand the QuickDraw GX concepts and data structures on which vectors are based. These concepts and data structures are introduced in the following sections.

A QuickTime vector corresponds to a QuickDraw GX path object, which is one of the geometric shapes supported by QuickDraw GX. Path objects are closely related to, and are in some cases defined by, other QuickDraw GX geometric shapes, such as points and curves.

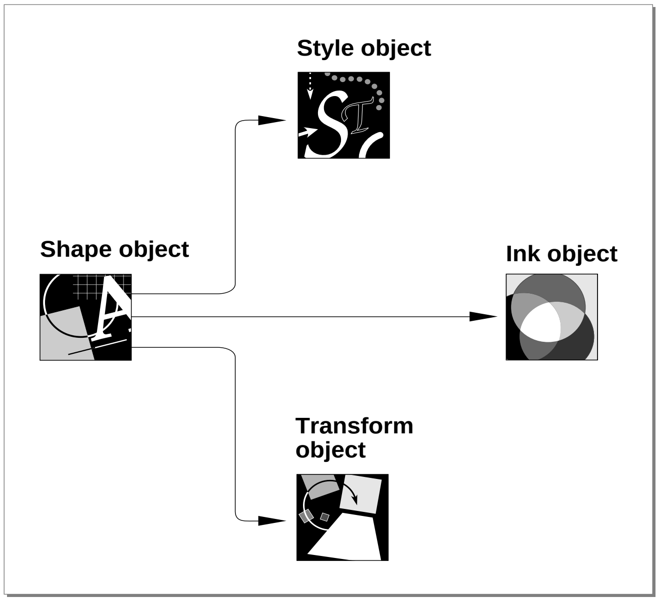
Shape Objects

A **shape** is something that you can draw. A shape consists of a **shape object** and three other associated objects (style, ink, and transform). A shape object consists of a **geometry** of a certain **shape type** (such as a line) and information about how the geometry is framed or filled when drawn. A shape also has attributes, such as whether it should be stored in accelerator-card memory, if present. It also has references to its other three related objects.

Supporting Objects

Several QuickDraw GX objects exist in support of shape objects. They are either directly or indirectly referenced by the shape object whose behavior they affect. Figure 24-1 shows the three objects that are directly referenced by a shape object.

Figure 24-1 A shape object and its referenced objects



Style Object

A **style object** describes certain characteristics affecting how a shape is drawn. For paths, this includes information such as the thickness of the pen, the joins between line segments, and any dash or pattern to apply to the shape.

Ink Object

An **ink object** describes a shape's color and its transfer mode—how that color is applied when the shape is drawn. Inks support many different kinds of color specification and many different transfer modes.

Attributes

Some objects have an attributes property, which is a group of flags that you use to modify the behavior of the object. In shapes, for example, these flags allow you to specify—among other things—how QuickDraw GX stores the shape object and how editing operations affect the shape object. In view ports, as another example, these flags allow you to specify behavior such as whether or not to perform color matching when drawing.

QuickDraw GX Coordinates

A **coordinate space** in QuickDraw GX consists of a plane in which positions are determined by coordinates. All coordinates in QuickDraw GX are specified with fixed-point numbers in the range of –32,768.0 to approximately 32,768.0. Coordinates are always written in the order (x, y), and for any coordinate space the point (0.0, 0.0) represents the origin of the space. Points that lie to the right of the origin increase in a positive direction along the x-axis; points that lie below the origin increase in a positive direction along the y-axis.

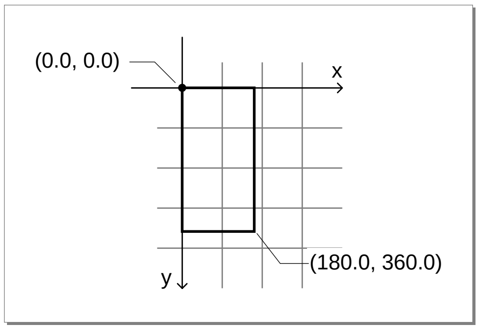
QuickDraw GX allows you to work in four coordinate spaces: geometry space, local space, global space, and device space. You can work separately in each space as appropriate; QuickDraw GX automatically converts among them when drawing. The spaces are described in order of their transformation during drawing.

Geometry Space

QuickDraw GX starts the drawing process by using the values in a shape's geometry. **Geometry space** is the space within which the fundamental position and dimensions of a shape object are defined. The numerical values in a shape's geometry define the shape's dimensions in geometry space. Suppose, for example, that the geometry of a rectangle consists of the points (0.0, 0.0) and (180.0, 360.0), as shown in Figure 24-2. In geometry space, the rectangle's origin is at (0.0, 0.0), its height is twice its width, and its area is 64,800.0 units square.

No distance metric, such as points per inch, is defined for geometry space. Thus, the absolute size of a shape is undefined in geometry space.

Figure 24-2 A rectangle in geometry space



Summary Table and Diagram of QuickDraw GX Objects

Three QuickDraw GX objects whose concepts are used for QuickTime vectors are summarized in Table 24-1.

Table 24-1 QuickDraw GX objects

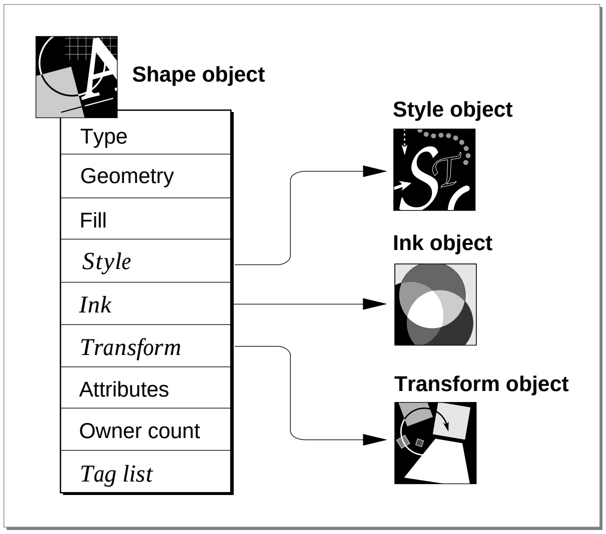
Object	Description
`Shape`	Defines the basic representation of a drawable entity. A shape object describes a geometry of a certain type (such as a path) and how the geometry is framed or filled when drawn. A shape also has references to its three related objects: style, ink, and transform.
`Style`	Describes certain characteristics affecting how a shape is drawn. For paths, this includes the thickness of the pen, the starting and ending caps for line segments, joins between line segments, and the dash or pattern to be applied to the shape. For typographic shapes, it includes the font, text size, and typeface of the text.

Table 24-1 QuickDraw GX objects (continued)

Object	Description
`Ink`	Describes a shape's color and its transfer mode (how the color is applied when the shape is drawn). Ink objects support many different kinds of color specification, and many different transfer modes.

Figure 24-3 shows the four objects used to represent a QuickDraw GX shape.

Figure 24-3 Basic components of a QuickDraw GX shape



Terminology Note
A QuickDraw GX *shape* is considered to be the combination of four objects just described. A *shape object* is one of the objects that make up the shape; it defines, among other characteristics, the shape's *geometry,* which is the description of the specific dimensions and location of the kind of shape that is to be drawn. ◆

Shape Properties

The properties of a shape object for the most part define the basic geometric characteristics of the shape. Shape objects have properties. Because a shape is an object and not a data structure, the order of the properties is completely arbitrary.

The fill property, which is used for paths, is a value that determines how a shape is filled or framed when drawn. QuickDraw GX provides a number of different ways of filling a shape. For example, a shape might have a **solid fill,** which indicates that the shape is solid—that is, the entire area enclosed by the sides of the rectangle is included in the shape. Alternatively, a shape might have a **framed fill,** which indicates that the shape is hollow. The section “Shape Fill” beginning on page 764 discusses types of shape fill.

Shape Type

The **shape type** property of a shape object specifies what type of shape the shape object represents. The type used for QuickTime vectors is the path, specified by the `gxPathType` constant. Its geometry includes any number of separate multiple-point path contours, each contour consisting of straight or curved line segments connecting its points.

Shape Geometry

For a path, the shape type does not contain all the geometric information necessary to define the shape. The geometries of paths contain (x, y) coordinate pairs called **geometric points**—points that specify the location, dimension, and form of the paths:

- Curve shapes store three geometric points in their geometry—one to specify where the curve starts, another to specify where the curve ends, and another, called the **off-curve control point,** to specify the tangents used to define the curve.
- A path geometry can also contain multiple contours, but each **path contour** can contain curves as well as straight lines.

There are two possible views of a geometry: as a list of `x, y` coordinate pairs and as geometric points plotted on a geometric grid. This second way of viewing geometries is used frequently throughout this chapter, as it shows not only the geometric points, but also the implied **edges** that connect them.

Typically, the figures in this chapter do not show the grid, but just the points and edges.

Each geometric point in a geometry has a **geometry index**—if you consider the geometry as a list of geometric points starting from the first geometric point of the first contour to the last geometric point of the last contour, the geometry index of a particular geometric point is its position in this list.

QuickDraw GX takes into consideration the direction that an edge is pointing in a number of circumstances:

- When filling a shape. QuickDraw GX allows you to choose how a shape should be filled. The next section, "Shape Fill," discusses how the direction of an edge can affect how QuickDraw GX fills a shape.
- When determining the **contour direction** of a contour.
- When determining the inside or outside of a contour. QuickDraw GX normally defines the right side of an edge to be the inside and the left side to be the outside.

Shape Fill

Each path object has a fill property. This property specifies how QuickDraw GX interprets the geometry of the shape: how the shape is drawn, how the shape is hit-tested, and how certain geometric operations, like intersection or union, interpret the shape. There are two basic types of shape fills:

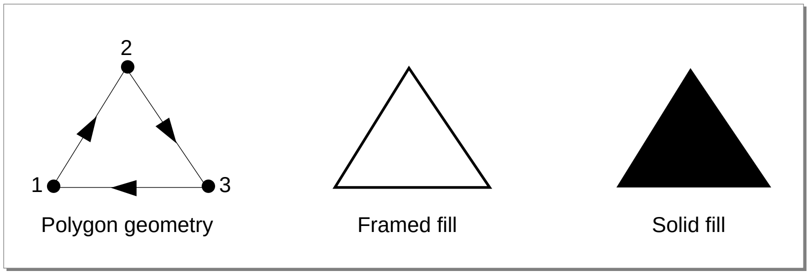
- **Framed fills.** These shape fills indicate that QuickDraw GX should interpret the shape as an outline—as a series of edges.
- **Solid fills.** These shape fills indicate that QuickDraw GX should interpret the shape as a solid area—the edges of the shape represent the boundaries of the area.

Note that framed fill, hollow fill, and solid fill are alternative names for open-frame fill, closed-frame fill, and even-odd fill, respectively, and that both inverse solid fill and inverse fill are alternate names for inverse even-odd fill.

A path can have any type of shape fill.

Figure 24-4 shows an example of a contour and how QuickDraw GX might draw it with a framed fill and with a solid fill.

Figure 24-4 Framed shapes versus solid shapes

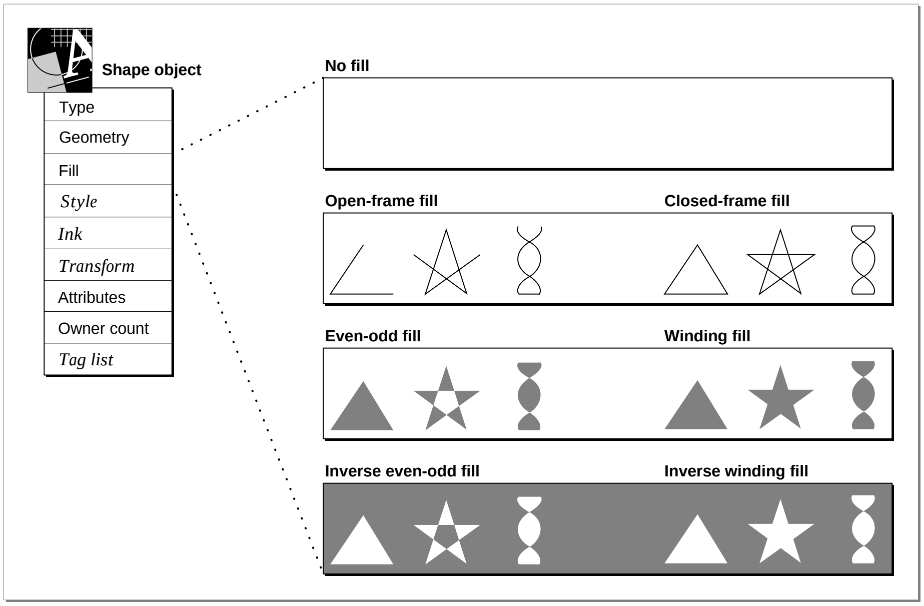


QuickDraw GX actually provides seven types of shape fills:

- no-fill shape fill
- open-frame shape fill (also called *frame fill*)
- closed-frame shape fill (also called *hollow fill*)
- even-odd shape fill (also called *solid fill*)
- winding shape fill
- inverse even-odd shape fill (also called *inverse fill* and *inverse solid fill*)
- inverse winding shape fill

Figure 24-5 shows these shape fills and the effect they have on three sample geometries.

Figure 24-5 The various shape fills and examples of their effects



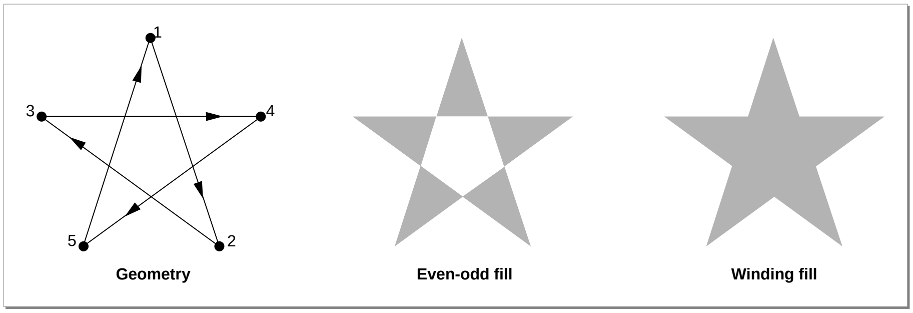
The no-fill shape fill specifies that QuickDraw GX should not draw the shape. You can use this shape fill to hide a shape. You can specify the no-fill shape fill for any shape type.

The open-frame shape fill specifies that QuickDraw GX should draw a shape as a connected set of edges. The closed-frame shape fill indicates that QuickDraw GX should also connect the last geometric point of a contour to the first geometric point of that contour.

The even-odd shape fill and the winding shape fill indicate that QuickDraw GX should interpret the shape as a solid area—the edges of the shape represent the boundaries of the area. These two shape fills differ in the algorithm they use to determine what area to include in the shape.

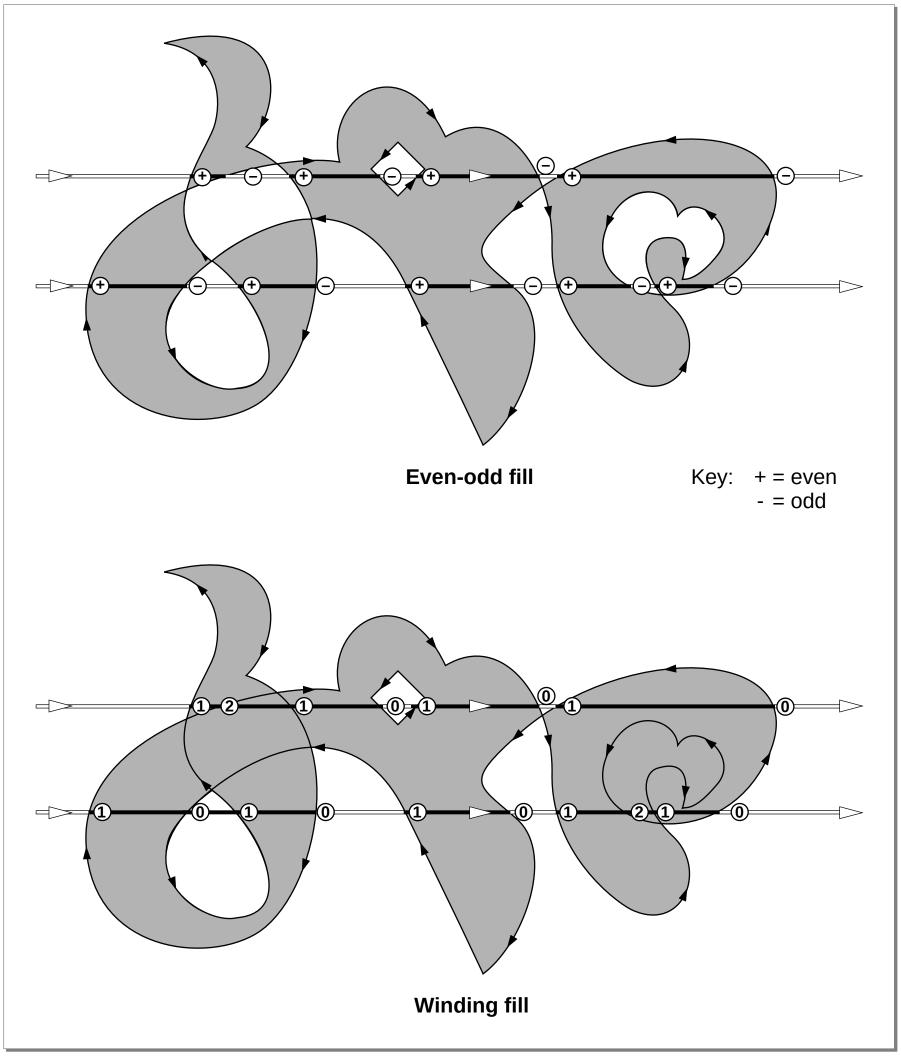
The even-odd shape fill indicates that QuickDraw GX should use the **even-odd rule** to determine what area lies inside a shape. As QuickDraw GX scans a shape horizontally, it fills the area between every other pair of edges, as shown in Figure 24-7.

Figure 24-6 Even-odd and winding fills



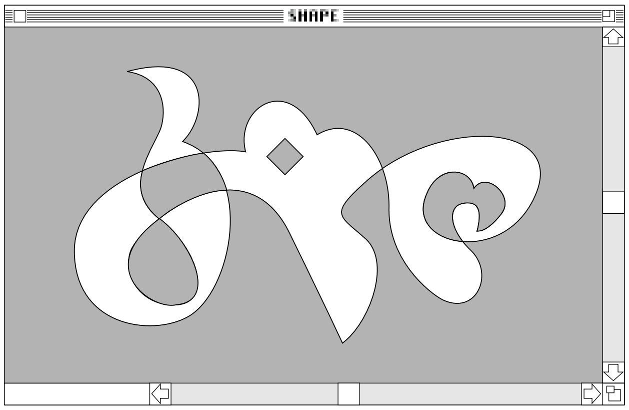
The winding shape fill indicates that QuickDraw GX should use the **winding-number rule** to determine what area lies inside a shape. As QuickDraw GX scans a shape horizontally, it increments a counter the first time it crosses an edge of the shape. It also notices whether the contour was directed up or down at that edge. As QuickDraw GX continues to scan the shape horizontally, every time it crosses another edge pointed in the same direction (up or down), it increments the counter, and when it crosses an edge pointing in the opposite direction (down or up), it decrements the counter. Wherever along the horizontal scan line the counter is not zero, QuickDraw GX fills the area, as is shown in Figure 24-7.

Figure 24-7 The even-odd rule and winding-number rule algorithms



The inverse even-odd shape fill indicates the inverse of the even-odd shape fill, as shown in Figure 24-8.

Figure 24-8 The inverse even-odd shape fill



Similarly, the inverse winding shape fill indicates the inverse of the winding shape fill.

The shape fill does more than affect the way a shape is drawn; it affects the fundamental behavior of a shape. Two shapes with the same geometry that have different shape fills can exhibit vastly different geometric behaviors. For example, the shape fill can affect

- stylistic variations
- shape measurements and other geometric operations.

Table 24-2 lists the defined constants for shape fill and describes what each one means. (Note that some shape fills have two or more equivalent names.) The constants are defined in the `gxShapeFills` enumeration.

Table 24-2 Shape fills

Constant	Value	Explanation
`gxNoFill`	0	No fill—the shape is not filled at all. QuickDraw GX does not draw a shape with this shape fill and you may not perform geometric operations on it. You can use this shape fill to temporarily hide shapes or to prevent parts of a picture from drawing.
`gxOpenFrameFill`	1	Open-frame fill—the shape is outlined instead of filled. With this shape fill, QuickDraw GX interprets the shape as a connected series of straight or curved lines from the starting point of the shape geometry to the ending point (but not back to the starting point again).
`gxFrameFill`	1	Framed fill (same as `gxOpenFrameFill`).
`gxClosedFrameFill`	2	Closed-frame fill—the shape is outlined instead of filled. As with the open-frame fill, QuickDraw GX interprets the shape as a series of lines (or curves) from the starting point of the shape geometry to the ending point. However, in this case, QuickDraw GX also includes a line (or curve) from the ending point to the starting point, thus closing the contour.
`gxHollowFill`	2	Hollow fill (same as `gxClosedFrameFill`).
`gxEvenOddFill`	3	Even-odd fill—the shape is filled using the even-odd rule. See Figure 24-6 for an illustration of this rule.
`gxSolidFill`	3	Solid fill (same as `gxEvenOddFill`).
`gxWindingFill`	4	Winding fill—the shape is filled using the winding-number rule. See Figure 24-6 (page 767) for an illustration of this rule.

Table 24-2 Shape fills (continued)

Constant	Value	Explanation
`gxInverseEvenOddFill`	5	Inverse even-odd fill—the shape is filled in an opposite manner from the even-odd rule; everything *not* filled using the even-odd rule is filled using this rule.
`gxInverseSolidFill`	5	Inverse solid fill (same as `gxInverseEvenOddFill`).
`gxInverseFill`	5	Inverse fill (same as `gxInverseEvenOddFill`).
`gxInverseWindingFill`	6	Inverse winding fill—the shape is filled using the winding-number rule and then inverted.

Path Shapes

A **path contour** is defined by a series of geometric points. However, a path contour can contain off-curve control points as well as on-curve points; therefore, a path contour can contain curves as well as straight lines. A **path shape** may include any number of path contours.

Note
A single path contour can have between 0 and 32,767 geometric points. The geometry of a path shape can between 0 and 32,767 contours. The total size of a path geometry may not exceed 2,147,483,647 bytes. ◆

Every path contains an array of **control bits** that specify which geometric points are on curve and which geometric points are off curve. QuickDraw GX connects two consecutive on-curve points with a straight line. If two on-curve points have an off-curve point between them, QuickDraw GX connects the two on-curve points with a quadratic Bezier curve, using the geometric point between them as the off-curve control point.

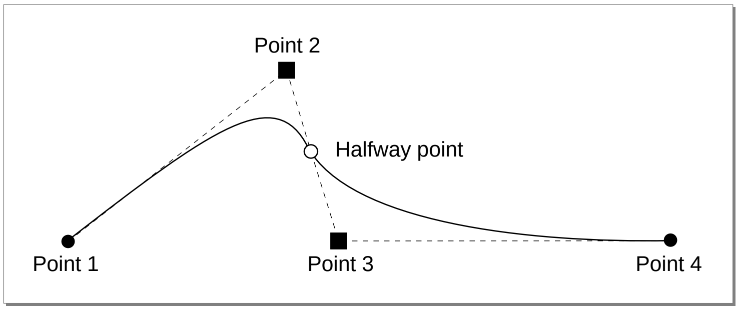
Quadratic Bezier curves have the following characteristics:

- A line connecting the first point and the off-curve control point describes the tangent of the curve at the first point.

- A line connecting the off-curve control point and the last point describes the tangent of the curve at the last point.
- The curve is always contained by the triangle formed by the three geometric points.
- The midpoint of the curve is halfway between the off-curve control point and the point midway between the first point and last point.

QuickDraw GX allows a path to have two or more consecutive off-curve control points. In this case, each pair of consecutive off-curve points implies an on-curve point midway between them, as represented by the small hollow circle in Figure 24-9.

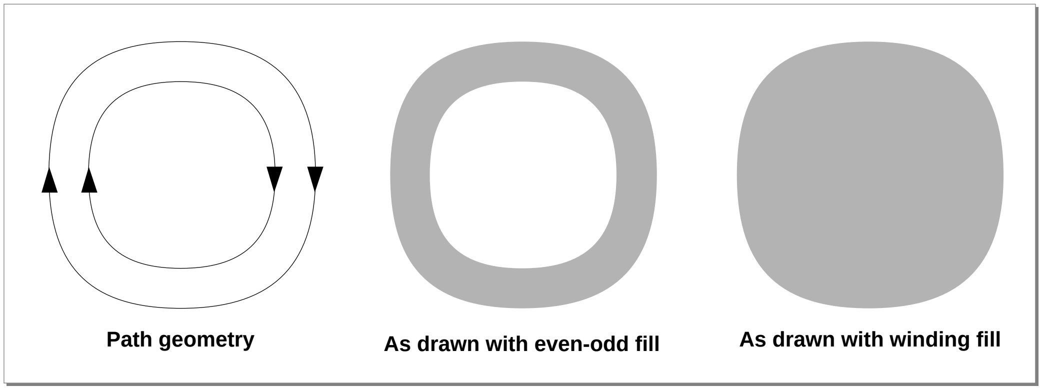
Figure 24-9 A path with two consecutive off-curve points



Path shapes may have any shape fill—including open-frame shape fill. However, a path may not have the open-frame shape fill if the first point or the last point of any path contour is an off-curve point.

If the contours of a path shape cross over one another, or if a path shape contains contours that lie within other contours, the even-odd shape fill and the winding shape fill may fill the path shape differently, as shown in Figure 24-10.

Figure 24-10 A path shape filled with the even-odd and winding shape fills



Contour direction affects how QuickDraw GX fills a path when the path has the winding shape fill. In the example in Figure 24-10, if the inner contour has the opposite contour direction from the outer contour, the winding shape fill works in the same manner as the even-odd shape fill.

When you create a new path shape, QuickDraw GX makes a copy of the default path shape. The default path shape has these properties:

- owner count: 1
- tag list: no tags
- shape attributes: no attributes
- shape type: path type
- shape fill: even-odd fill
- geometry: 0 contours, 0 points

You may change the properties of the default path shape. However, when creating a new path shape, QuickDraw GX always initializes the owner count to 1 and the geometry to 0 contours with 0 points, even if you have specified other values for the default paths shape.

For examples of creating and drawing path shapes without stylistic variations, see “Creating and Drawing Paths” beginning on page 774.

Creating and Drawing Paths

A path contour is a series of connected points. The contour can contain both straight lines and curves. Therefore, the geometric points that make up a path contour can be on-curve points or off-curve control points. QuickDraw GX defines the `gxPath` structure to encapsulate a path contour geometry:

```
struct gxPath {
    long                vectors;
    long                controlBits[gxAnyNumber];
    struct gxPoint      vector[gxAnyNumber];
};
```

The `vectors` field indicates the number of geometric points in the path and the `vector` array contains the geometric points themselves. The `controlBits` array specifies which geometric points are on-curve points and which are off-curve control points. A value of 0 indicates an on-curve point and a value of 1 indicates an off-curve point. For example, a `controlBits` field with the value

```
0x55555555   /* 0101 0101 0101 0101 ... */
```

indicates that every other point is an off-curve control point; the first point is on curve, the second point is off, and so on. As another example, a `controlBits` field value of

```
0x00000000   /* 0000 0000 0000 0000 ... */
```

indicates all points are on curve, which effectively creates a polygon.

Notice that the `controlBits` array allows you to specify sequential off-curve control points. For example, a `controlBits` value of

```
0xFFFFFFFF   /* 1111 1111 1111 1111 ... */
```

indicates that all points are off curve. When you indicate that two control points in a row are off curve, QuickDraw GX assumes an on-curve point midway between them.

The path shape allows you to group any number of contours within a single QuickDraw GX shape. The `gxPaths` structure encapsulates the multiple-path geometry:

```
struct gxPaths {
    long            contours;
    struct gxPath   contour[gxAnyNumber];
};
```

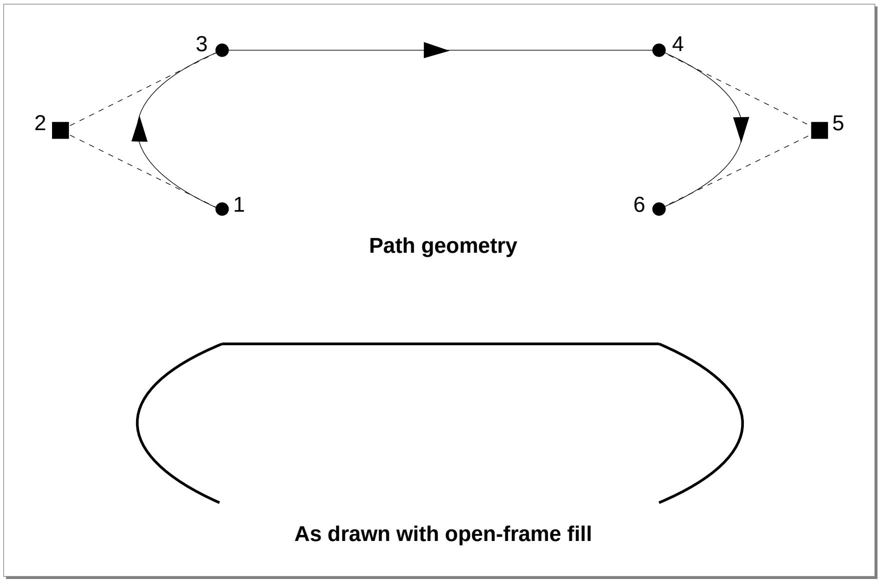
The `contours` field indicates the total number of contours (in other words, the total number of separate paths), and the `contour` array contains the path geometries.

Creating Paths With a Single Contour

Since a `gxPaths` structure is of variable length and every element in it is of type `long`, you can define a path geometry as an array of long integer values.

The path shown in Figure 24-11 has four on-curve points and two off-curve points. When drawn with the open-frame shape fill, it contains two curves and one straight line.

Figure 24-11 A path



For more information about shape fills, see “Shape Fill” (page 764).

Creating Paths Using Only Off-Curve Points

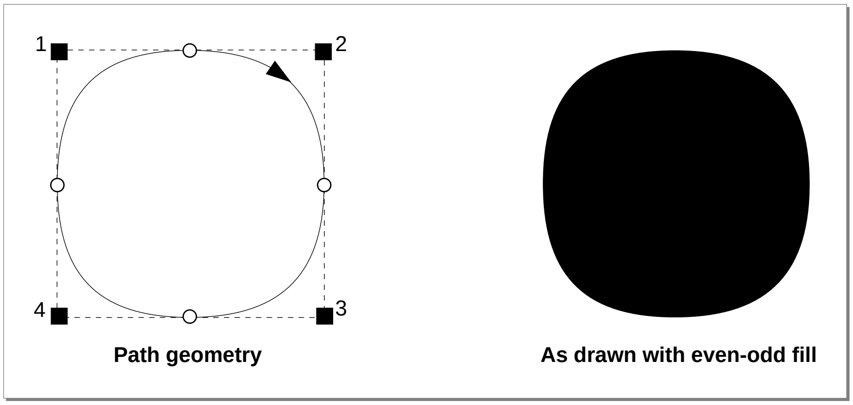
The next example shows how you can create a path using only off-curve control points. The path defined in this example contains four control points, and the `controlBits` field is set to

```
0xF0000000    /* 1111 0000 0000 0000 0000 ... */
```

which indicates that the first four points are off curve. The path contains only four points, and therefore they are all off curve.

The four off-curve control points in this example form a square; the path that they define is a rounded square, as shown in Figure 24-12.

Figure 24-12 A round path shape



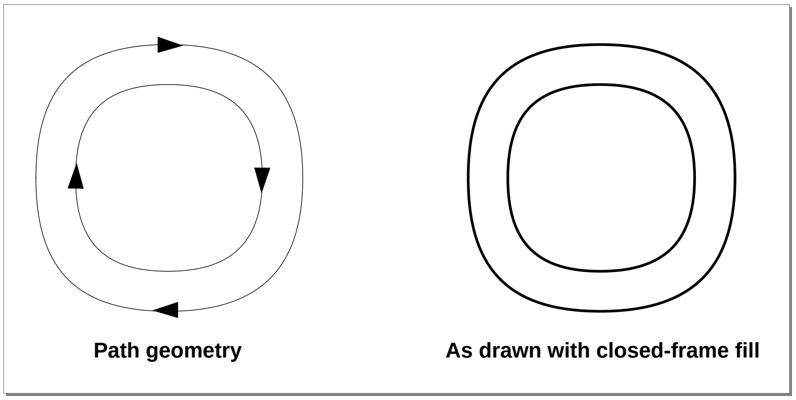
Notice that the path is filled with the even-odd shape fill, which is the default for path shapes. You could, however, specify any shape fill for this path except the open-frame shape fill. The open-frame shape fill requires that the first and last points of the contour be on-curve points, and this path has no on-curve points.

Creating Paths With Multiple Contours

The next example shows how a single path shape can contain more than one path contour. The path shape defined in this example includes the round path from the previous example as well as a second round path, entirely contained within the first.

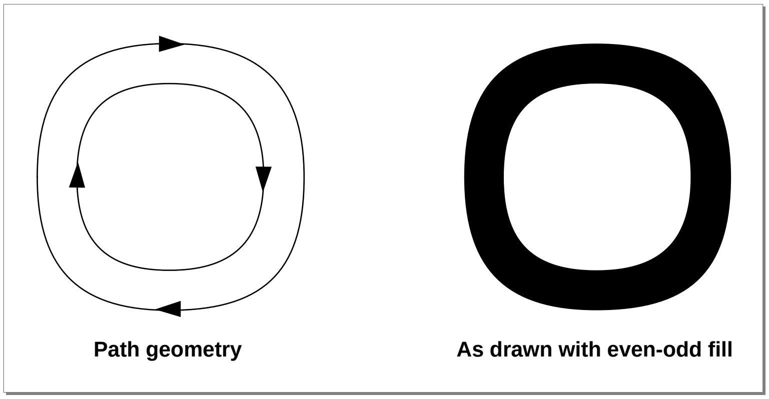
This is shown in Figure 24-13.

Figure 24-13 A path shape with two concentric clockwise contours and closed-frame shape fill



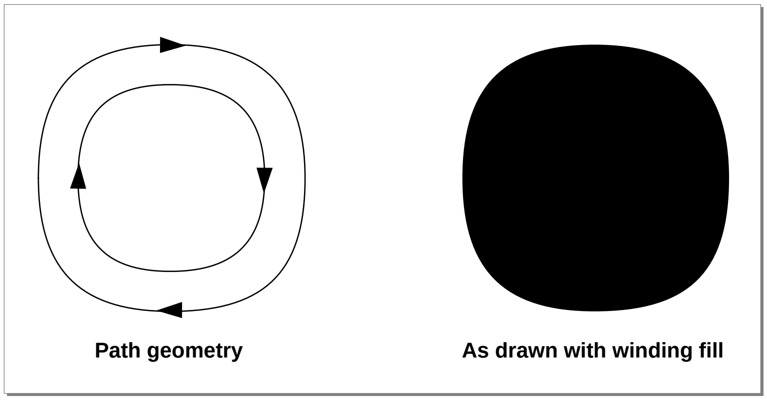
If you don't specify a shape fill, you get the shape fill from the default path shape, which is the even-odd shape fill. The path shape resulting from an even-odd shape fill is shown in Figure 24-14.

Figure 24-14 Two concentric clockwise contours and even-odd shape fill



Notice that the even-odd shape fill causes the fill of the outer contour, but not the inner contour. However, if you specify the winding shape fill for this path, the resulting shape would appear as shown in Figure 24-15.

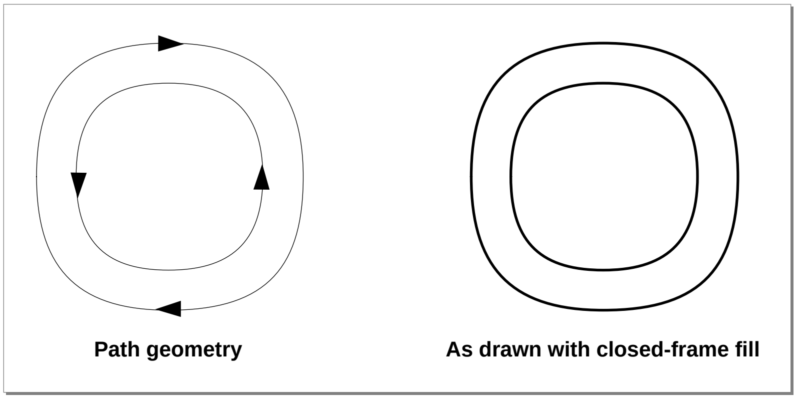
Figure 24-15 Two concentric clockwise contours and winding shape fill



Unlike the even-odd shape fill, the winding shape fill causes QuickDraw GX to fill inner contours—*as long as the inner contour has the same contour direction as the* *outer contour*. If the inner contour and the outer contour have opposite contour directions, neither the even-odd shape fill nor the winding shape fill will fill the inner contour.

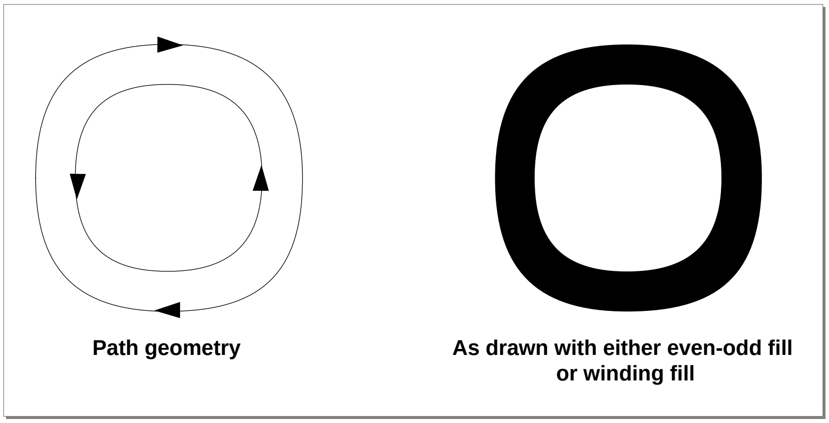
For example, if you change the direction of the inner contour from the previous example by reversing the order of the second path's geometric points and set the shape fill to the closed-frame shape fill, the resulting shape has contours with opposite contour directions, as depicted in Figure 24-16.

Figure 24-16 Internal counterclockwise contour and closed-frame shape fill



Since the outer contour and the inner contour have opposite contour directions, neither the even-odd shape fill nor the winding shape fill cause QuickDraw GX to fill the inner contour, as shown in Figure 24-17.

Figure 24-17 A path shape with even-odd or winding shape fill



Geometric Properties of Style Objects

The following sections describe the geometric properties of style objects, which you can use to apply certain types of stylistic variations to QuickDraw GX shapes. In particular, they show how you can

- specify the pen width to use when drawing a shape's frame
- indicate the placement of the pen relative to the shape's frame
- specify what to draw at the corners of a shape's contours
- fill a shape, or the frame of a shapes

About Geometric Styles

A style is a group of stylistic variations applied to a shape. QuickDraw GX provides two major categories of stylistic variations: geometric variations, which include pen width, dashes, patterns, and so on, and typographic variations, which include font, font size, type style, and so on.

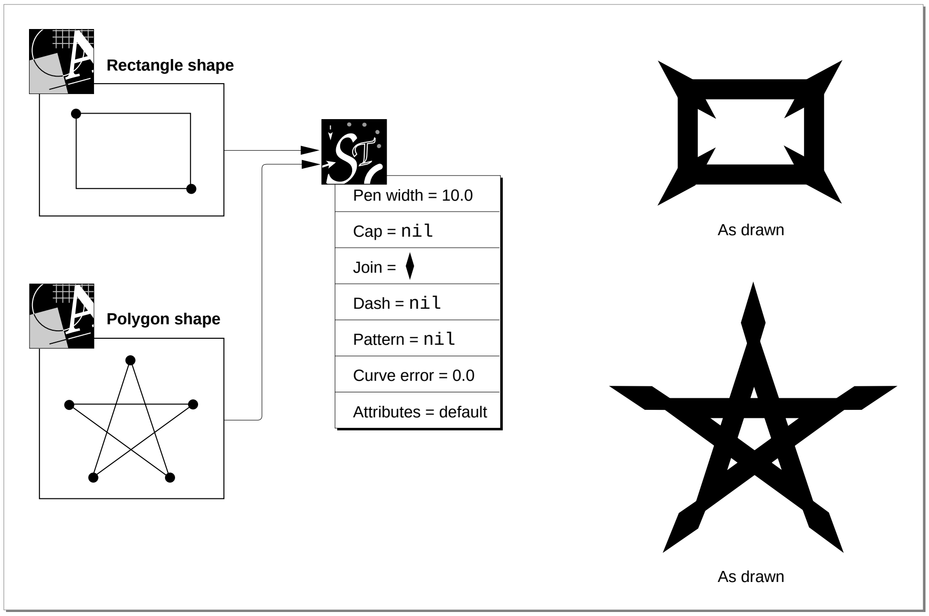
Both types of stylistic variation are encapsulated in a **style object.** Like a shape object, a style object is a data structure. Each style object has a group of properties, and each **style property** represents a different stylistic variation.

Shapes and Styles

In general, a shape object is an object with a group of properties that describe a geometry; a style object is an object with a group of properties that affect how QuickDraw GX interprets a shape's geometry during drawing.

Every QuickDraw GX shape object contains a reference to a style object. As Figure 24-18 depicts, a single style object can be shared by multiple shape objects.

Figure 24-18 Shared style objects



As with all objects, a style object has an owner count, which reflects the number of existing references to the style object. When a new reference to a style object is created, the owner count of the style object is incremented; when a reference to a style object goes away, the owner count of the style object is decremented. When a style object has an owner count of 0, QuickDraw GX can free the memory used by the style object.

References to style objects typically include those contained in shape objects and those contained in variables in your application. QuickDraw GX manages the owner counts corresponding to references in shape objects for you; you are responsible for managing the owner counts corresponding to variables in your application.

Style Properties

A style object contains properties that the drawing of a shape. The style properties that affect paths are:

- **Pen width.** This property specifies the width of the pen QuickDraw GX uses to draw the contours of a shape.
- **Style attributes.** This property is a group of flags that allow you to specify how QuickDraw GX places the pen with respect to a shape's geometry and whether the shape should be constrained to a grid when drawn.
- **Join.** This property specifies what QuickDraw GX should draw at the corners of a shape's contours. QuickDraw GX provides two standard join types (one for round corners and one for sharp corners).

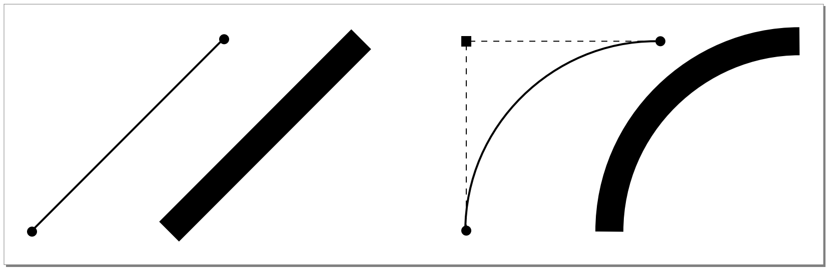
The Geometric Pen

The contours of framed paths are drawn with the QuickDraw GX **geometric pen.** You can specify the width of this pen using the pen width property of the style object, and you can specify where to place the pen relative to the contours of a shape using the style attributes, which are described in "Style Attributes" beginning on page 783.

Conceptually, the QuickDraw GX geometric pen is a line that QuickDraw GX drags along the contours of the shape being drawn—always keeping it perpendicular to the contours. In effect, the geometric pen turns a framed geometry into a filled one. For example, a line shape, which is always framed, becomes the equivalent of a filled polygon after QuickDraw GX applies the geometric pen.

Figure 24-19 shows the effect of the geometric pen. This figure shows two geometries— a line geometry and a curve geometry—and how QuickDraw GX draws them with a pen width of 15.

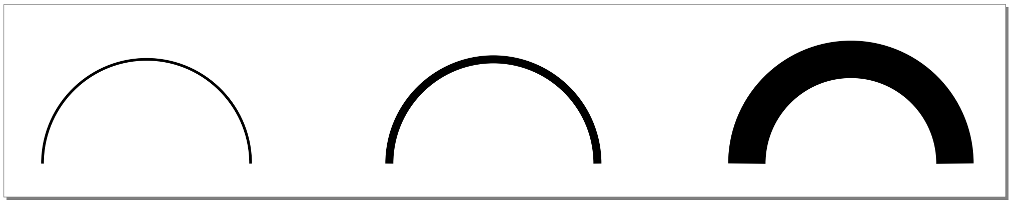
Figure 24-19 The QuickDraw GX geometric pen



Notice that the ends of the thick line contour and the thick curve contour in Figure 24-19 are perpendicular to the direction of the contours themselves.

Figure 24-20 shows the effect of different pen widths on a semicircular path shape.

Figure 24-20 Differing pen widths



Style Attributes

The **style attributes** property of a style object contains six attributes that affect the drawing of a shape. Four of these attributes affect how QuickDraw GX places the geometric pen relative to the contours of a shape:

- The center-frame style attribute, which is the default, indicates that the QuickDraw GX should center the geometric pen along the shape's contours.
- The inside-frame style attribute indicates that QuickDraw GX should position the pen along the inside of a shape's contours.

- The outside-frame style attribute indicates that QuickDraw GX should position the pen along the outside of shape's contours.
- The auto-inset style attribute affects the definition of the inside and outside of a contour.

These four attributes are discussed in the next section, "Pen Placement."

There are also two style attributes that determine whether the geometric points of a shape are constrained to a grid when the shape is drawn:

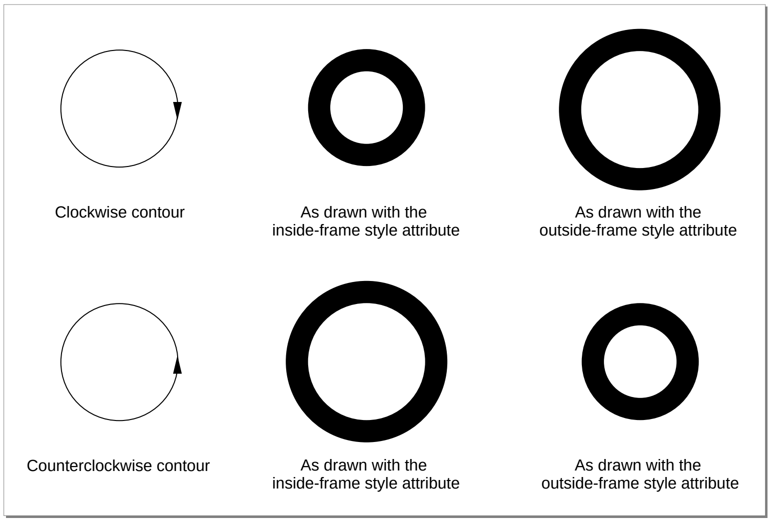
- The source-grid style attribute constrains the geometric points of a shape to integer values before applying the shape's style and transform information.
- The device-grid style attribute constrains the geometric points of a shape to integer pixel positions after applying the shape's style and transform information.

Pen Placement

You can use the center-frame, inside-frame, and outside-frame style attributes to specify where QuickDraw GX should position the pen with respect to the shape's geometry.

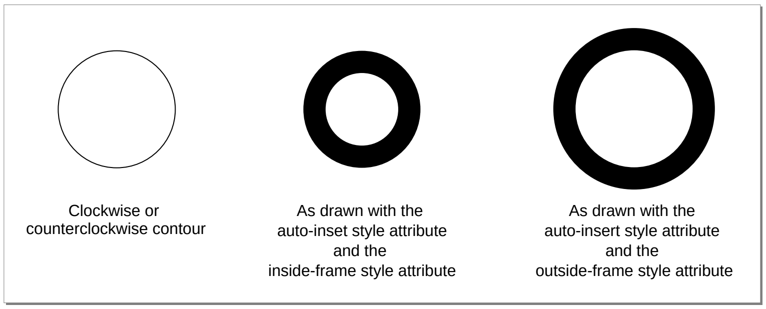
Figure 24-21 shows the results of these style attributes. Notice that QuickDraw GX considers contour direction when determining which side of a contour is the inside: the right side of the contour is the inside, while the left side of the contour is the outside.

Figure 24-21 Pen placement


Clockwise contour
As drawn with the inside-frame style attribute
As drawn with the outside-frame style attribute
Counterclockwise contour
As drawn with the inside-frame style attribute
As drawn with the outside-frame style attribute

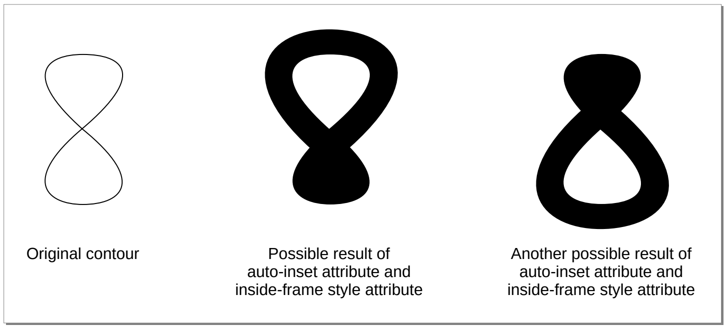
QuickDraw GX also provides the auto-inset style attribute, which allows you to specify that QuickDraw GX should ignore contour direction when determining which side of a contour is the inside. When you set this style attribute, QuickDraw GX determines the true inside of a contour, rather than using the right side as the inside. Figure 24-22 shows the effect of setting the auto-inset style attribute for the shapes depicted in Figure 24-21.

Figure 24-22 Effect of the auto-inset style attribute


Clockwise or counterclockwise contour
As drawn with the auto-inset style attribute and the inside-frame style attribute
As drawn with the auto-insert style attribute and the outside-frame style attribute

When a contour crosses over itself, the results of setting the auto-inset style attribute are unpredictable, as the contour has no true inside (or, actually, has multiple true insides). For the figure-eight shape in Figure 24-23, setting the `gxAutoInsetStyle` and the `gxInsideFrameStyle` style attributes could lead to one of two results.

Figure 24-23 Effect of the auto-inset and inside-frame style attributes for a crossed contour

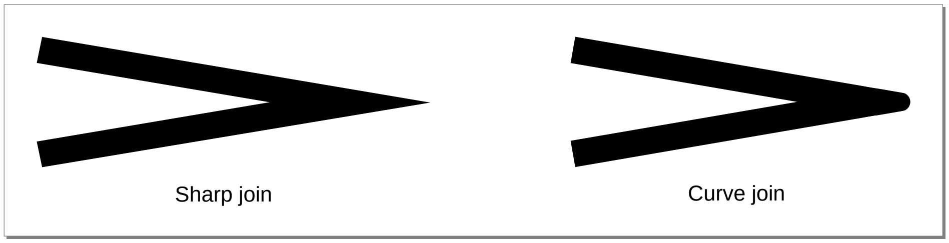

Original contour
Possible result of auto-inset attribute and inside-frame style attribute
Another possible result of auto-inset attribute and inside-frame style attribute

Joins

QuickDraw GX uses the **join property** of a shape's style object to store information about the joins of a shape.

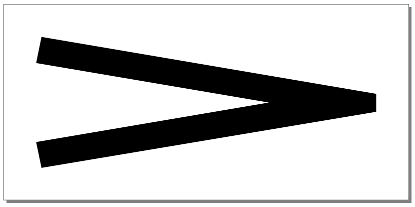
You can use the join attributes to specify two types of standard joins—sharp joins and curve joins, as shown in Figure 24-24.

Figure 24-24 Standard joins



For sharp joins, QuickDraw GX allows you to specify a **miter**—the maximum distance between the actual corner of a shape's geometry and the corner as drawn, as shown in Figure 24-25.

Figure 24-25 Sharp join with miter



Transfer Modes

An **ink object** describes a shape's color and its transfer mode—how that color is applied when the shape is drawn. Inks support many different kinds of color specification, and many different transfer modes.

This section explains the use of transfer modes in drawing shapes. It also introduces QuickDraw GX ink objects and describes their properties. Finally, it describes how transfer modes work in QuickDraw GX.

About Ink Objects

An ink object exists to provide color information about a shape. Each QuickDraw GX shape consists of a shape object, a style object, an ink object, and a transform object; the ink object associated with a shape defines the color with which the shape is drawn, as well as the transfer mode used to draw it.

Inks are device independent. Their information is not affected by the properties of the display device to which the shapes they modify are drawn. When it draws a shape on a device, QuickDraw GX approximates as closely as possible the color specified by the shape's ink.

Ink Properties

For QuickTime vectors, there is one accessible property in an ink object. **Transfer mode** is the way, or mode, of transferring the color to its destination (the screen or printed page or other location into which the shape associated with this ink is drawn). Transfer mode is a specification (such as “copy” or “XOR” or “blend”) of the interaction between the color in this ink object and the existing color or colors of the destination. With transfer mode you can make a shape opaque or transparent, draw only part of it, change its color, or combine its color with the destination color in many different ways. The transfer mode also includes a specification of a color space.

Color

One main purpose of an ink object’s existence is to specify the color of a shape. Because there is only one ink object per shape, it follows that each QuickDraw GX shape can have only one color. The only exception to this is for bitmap shapes, which use pixel values rather than an ink object to specify colors. (Picture shapes have no color at all apart from the colors of their component shapes, and thus do not use their ink object.)

The color in an ink object is defined with a `gxColor` structure:

```
struct gxColor{
    gxColorSpace                space;
    gxColorProfile              profile;
    union {
            struct gxCMYKColor              cmyk;
            struct gxRGBColor               rgb;
            struct gxRGBAColor              rgba;
            struct gxHSVColor               hsv;
            struct gxHLSColor               hls;
            struct gxXYZColor               xyz;
            struct gxYXYColor               yxy;
            struct gxLUVColor               luv;
            struct gxLABColor               lab;
            struct gxYIQColor               yiq;
            gxColorValue                    gray;
            struct gxGrayAColor             graya;
            unsigned short                  pixel16;
            unsigned long                   pixel32;
            struct gxIndexedColor           indexed;
            gxColorValue                    component[4];
    } element;
};
```

The color structure specifies three characteristics of a color:

- The color's color space, which tells what kind of format the color has—such as red-green-blue (RGB), hue-saturation-value (HSV), or luminance (grayscale).
- For QuickTime vectors, the reference to a color profile object must be `nil`.
- The numeric color values that (for the given color space) specify the color of this ink object. An individual color has one number for each dimension, or color component, in the color's color space; for example, an RGB color value consists of three color component values. A color may consist of a maximum of four components.

Transfer Mode

The transfer mode in an ink object is contained in a `gxTransferMode` structure:

```
struct gxTransferMode{
    gxColorSpace                    space;
    gxColorSet                      set;
    gxColorProfile                  profile;
    Fixed                           sourceMatrix[5][4];
    Fixed                           deviceMatrix[5][4];
    Fixed                           resultMatrix[5][4];
    gxTransferFlag                  flags;
    struct gxTransferComponent      component[4];
};
```

Like the color structure just described, the transfer mode structure specifies a color space. A transfer mode specifies its own color space because it can perform its operations according to its own definitions of color, independent of the color specifications in the rest of the ink object.

The transfer mode structure contains three 5×4 matrices (5 rows, 4 columns), the **source matrix, device matrix,** and **result matrix,** which it can use to transform colors for special effects, by blending proportions of the colors' components. In addition, it contains a set of **transfer mode flags** that control several aspects of the transfer mode operation.

The structure also contains up to four **transfer components,** used along with the matrices in the transfer mode operation. Transfer components contain the actual specification of the mode of transfer to use when drawing. Transfer components are defined by the `gxTransferComponent` structure:

```
struct gxTransferComponent{
        gxComponentMode         mode;
        gxComponentFlag         flags;
        gxColorValue            sourceMinimum;
        gxColorValue            sourceMaximum;
        gxColorValue            deviceMinimum;
        gxColorValue            deviceMaximum;
        gxColorValue            clampMinimum;
        gxColorValue            clampMaximum;
        gxColorValue            operand;
};
```

A transfer component contains a **component mode** specifying the type of transfer mode (like “copy” or “XOR”) to use, an operand to apply (if the type calls for an operand), a set of maximum and minimum color values, and a set of flags. There is one transfer component for each color component (dimension) in the transfer mode’s color space. Each of the transfer components in the transfer mode structure may specify a different component mode, which means that each dimension of a color space can be drawn with a different transfer mode when a shape is drawn.

How these parts of the transfer mode structure and transfer component structure define the transfer mode for drawing, and how you can use transfer modes to obtain the proper effect when drawing, are described in the section “Transfer Modes” beginning on page 814.

Color in QuickDraw GX

In QuickDraw GX, color information about a shape is kept in the ink object associated with the shape object. A shape’s ink object describes both the color of the shape and the transfer mode with which the shape is drawn.

Note
For QuickTime vectors, color information is used for transfer modes. These are described in “Transfer Modes” (page 814). ◆

QuickDraw GX has a powerful, device-independent method for representing color in many different formats. Conversion among the formats is simple and direct, and in many cases automatic. QuickDraw GX also provides automatic manipulation of device-specific colors so that colors match consistently when scanned from or drawn to many different imaging devices.

This section describes how color is represented and how you can manipulate color information. It presents the information in this order:

- Colors are numerical values that make sense only in terms of specific color spaces. Color spaces are described first, under “Color Spaces” (page 791).
- The mathematical values used by each color space are combined with other information to make a color structure. How color values relate to the color structure is described second, under “Color-Component Values, Color Values, and Colors” (page 812).

Color Spaces

A **color space** specifies how color information is represented. It defines a one-, two-, three-, or four-dimensional space whose dimensions, or **components,** represent intensity values. For example, RGB space is a three-dimensional color space whose components are the red, green, and blue intensities that make up a given color. Visually, these spaces are often represented by various solid shapes, such as cubes, cones, or polyhedra.

QuickDraw GX directly supports 28 different color spaces, to give you the convenience of working in whatever kinds of color data most suits your needs. The QuickDraw GX color spaces fall into several groups, or **base families.** They are

- luminance-based color spaces, used for grayscale display and printing
- RGB-based color spaces, used mainly for color video display
- CMYK-based color spaces, used mainly for color printing
- universal color spaces, used mainly for device-independent color measurements

All color spaces within a base family differ only in details of storage format or else are related to each other by very simple mathematical formulas.

Within a base family, some of the differences among color spaces relate to their **packing,** the number of bits used to store each color component. For example, RGB colors might be stored with 5, 8, or 16 bits per component. Each storage format is a different color space. Internally, QuickDraw GX always converts colors so that each component has 16 bits; thus you can think of the

16-bit-per-component color spaces as the fundamental ones in each base family, and those with smaller storage spaces as packed (storage-compressed) versions.

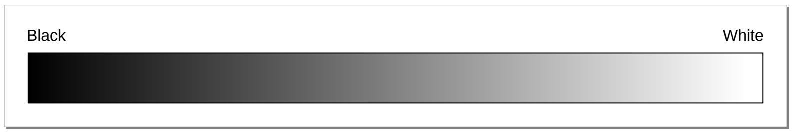
Some QuickDraw GX color spaces have an alpha channel, an additional component that measures opacity or transparency. Alpha channels are described in the section “Color Spaces With Alpha Channels” (page 811).

The `gxColorSpaces` enumeration lists the color spaces directly supported by QuickDraw GX. Each color space has its own format for representing color information. The rest of this section discusses those color spaces and their formats.

Luminance-Based Color Spaces

Luminance is a scale of lightness. Luminance-based color spaces, or gray spaces, typically have a single component, ranging from black to white, as shown in Figure 24-26. Luminance-based color spaces are used for black-and white and grayscale display and printing.

Figure 24-26 Luminance color space



A color is converted into luminance by evaluating its overall lightness. The luminance of a color expressed in RGB (see “RGB-Based Color Spaces” (page 794)), for example, can be calculated approximately with this formula:

```
luminance = 0.30 * red  + 0.59 * green + 0.11 * blue;
```

The luminance-based color spaces supported by QuickDraw GX (and defined in the `gxColorSpaces` enumeration) are `gxGraySpace` and `gxGrayASpace`. The *A* in `gxGrayASpace` stands for a second component called an *alpha channel;* see the section “Color Spaces With Alpha Channels” (page 811) for more information.

Table 24-3 describes details of the storage formats for `gxGraySpace` and `gxGrayASpace`. In each of these spaces, the luminance is specified by a single number whose range varies from 0 to 65,535. The color black has a luminance value of 0, regardless of the color space.

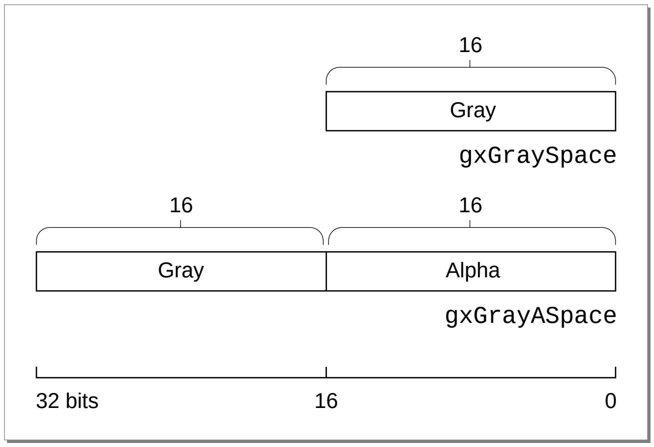
Table 24-3 Luminance-based color spaces supported by QuickDraw GX

Constant	Enumeration Value	Explanation
`gxGraySpace`	0x000A	16 bits per component (gray only); component values range from 0 to 0xFFFF. Total storage size for each color value: 16 bits.
`gxGrayASpace`	0x008A	16 bits per component (gray and alpha); component values range from 0 to 0xFFFF. Total storage size for each color value: 32 bits. Alpha channels are described on (page 811).

Figure 24-27 is a visual representation of the storage formats for the luminance-based color spaces.

Note
This figure and all subsequent storage-format figures in this chapter assume that data storage is “big-endian,” that is, that lower addresses correspond to higher-order bytes in a word or long word value. For processors whose storage model is different, the elements of the figures would be in a different order. These figures are presented for illustrative purposes only, and are not intended to specify details of storage order. ◆

Figure 24-27 Storage formats for luminance-based color spaces



RGB-Based Color Spaces

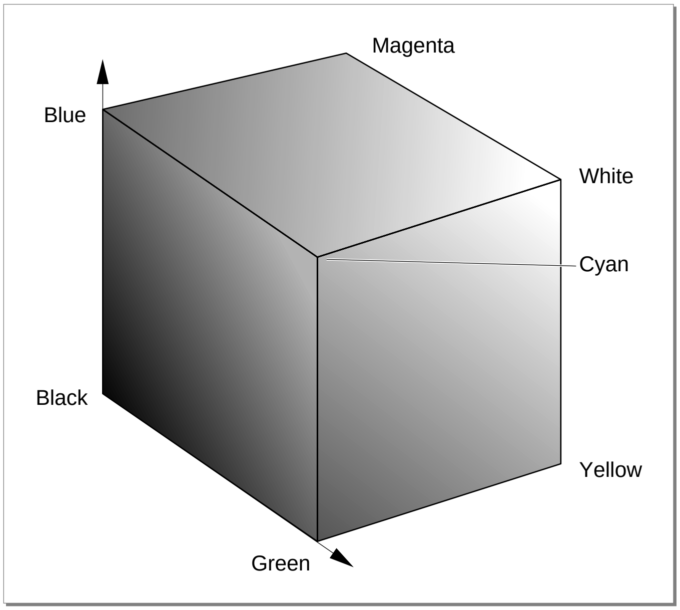
RGB-based color spaces are the most commonly used color spaces in computer graphics, primarily because they are directly supported by most color monitors. The groups of color spaces within the RGB base family include

- RGB spaces
- HSV and HLS spaces

RGB Spaces

Any color expressed in **RGB space** is some mixture of three primary colors red, green, and blue. Most RGB-based color spaces can be visualized as a cube, as in Figure 24-28, with corners of black, the three primaries (red, green, and blue), the three secondaries (cyan, magenta, and yellow), and white.

Figure 24-28 RGB color space

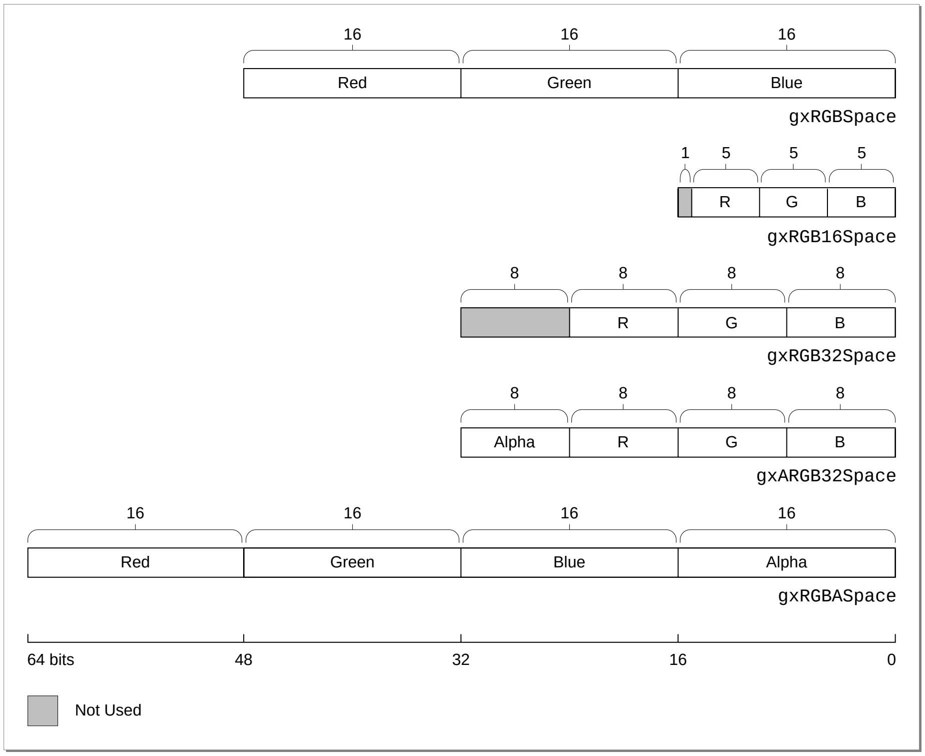


The RGB color spaces supported by QuickDraw GX (and defined in the `gxColorSpaces` enumeration) are `gxRGBSpace`, `gxRGB16Space`, `gxRGB32Space`, `gxRGBASpace`, and `gxARGB32Space`. See Table 24-4 and Figure 24-29 for storage-format details. In each of these spaces, a color value is represented by three or four color components, representing red, green, blue (and in some cases alpha); each component can vary in the number of bits used for its storage. The color black is represented by component values of 0 in the red, green, and blue components.

Table 24-4 RGB color spaces supported by QuickDraw GX

Constant	Enumeration Value	Explanation
`gxRGBSpace`	0x0001	16 bits per component (red, green, and blue); component values range from 0 to 0xFFFF. Total storage size for each color value: 48 bits.
`gxRGB16Space`	0x0501	5 bits per component (red, green, and blue); component values range from 0 to 0x1F. Total storage size for each color value: 16 bits (bit 15 is not used).
`gxRGB32Space`	0x0801	8 bits per component (red, green, and blue); component values range from 0 to 0xFF. Total storage size for each color value: 32 bits (bits 24–31 are not used).
`gxARGB32Space`	0x1881	8 bits per component (red, green, blue, and alpha); component values range from 0 to 0xFF. Total storage size for each color value: 32 bits. Alpha channels are described on (page 811).
`gxRGBASpace`	0x0081	16 bits per component (red, green, blue, and alpha); component values range from 0 to 0xFFFF. Total storage size for each color value: 64 bits. Alpha channels are described on (page 811).

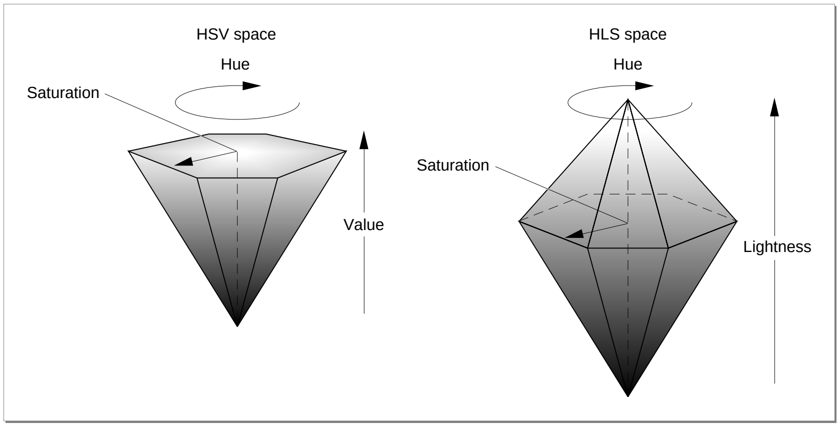
Figure 24-29 Storage formats for RGB color spaces



HSV and HLS Color Spaces

HSV space and **HLS space** are transformations of RGB space that allow colors to be described in terms more natural to an artist. The name *HSV* stands for *hue, saturation,* and *value,* and *HLS* stands for *hue, lightness,* and *saturation*. The two spaces can be thought of as being single and double cones, as shown in Figure 24-30.

Figure 24-30 HSV color space and HLS color space



The components in HLS space are analogous, but not completely identical, to the components in HSV space:

- The hue component in both color spaces is an angular measurement, analogous to position around a color wheel. A hue value of 0 indicates the color red; the color green is at a value corresponding to 120°, and the color blue is at a value corresponding to 240°. Horizontal planes through the cones in Figure 24-30 are hexagons; the primaries and secondaries (red, yellow, green, cyan, blue, and magenta) occur at the vertices of the hexagons.
- The saturation component in both color spaces describes color intensity. A saturation value of 0 (in the middle of a hexagon) means that the color is “colorless” (gray); a saturation value at the maximum (at the outer edge of a hexagon) means that the color is at maximum “colorfulness” for that hue angle and brightness.
- The value component (in HSV space) and the lightness component (in HLS space) describe brightness or luminance. In both color spaces, a value of 0 represents black. In HSV space, a maximum value for value means that the color is at its brightest. In HLS space, a maximum value for lightness means that the color is white, regardless of the current values of the hue and saturation components. The brightest, most intense color in HLS space occurs at a lightness value of exactly half the maximum.

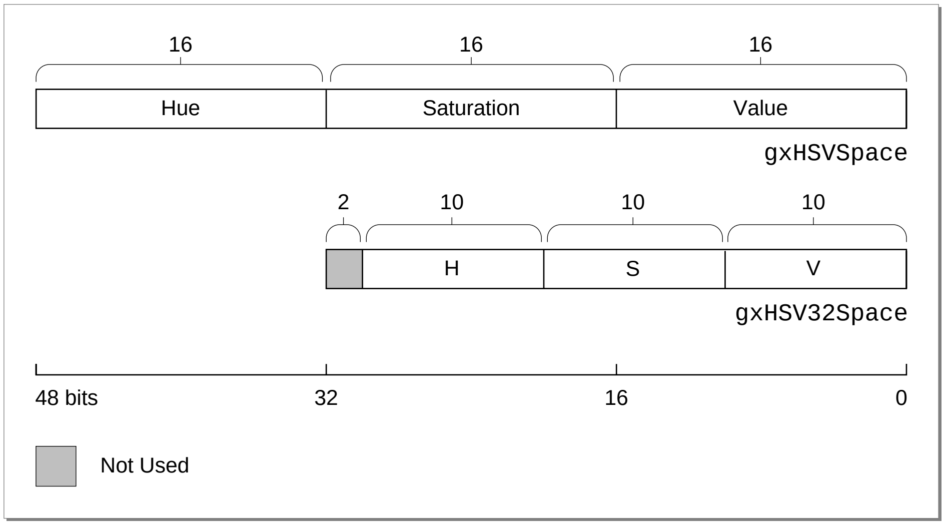
The HLS and HSV color spaces supported by QuickDraw GX (and defined in the `gxColorSpaces` enumeration) are `gxHSVSpace`, `gxHLSSpace`, `gxHSV32Space`, and `gxHLS32Space`. See Table 24-5 and Figure 24-31 for details of storage format.

Table 24-5 HSV and HLS color spaces supported by QuickDraw GX

Constant	Enumeration Value	Explanation
`gxHSVSpace`	0x0003	16 bits per component (hue, saturation, and value); component values range from 0 to 0xFFFF. Total storage size for each color value: 48 bits.
`gxHLSSpace`	0x0004	16 bits per component (hue, lightness, and saturation); component values range from 0 to 0xFFFF. Total storage size for each color value: 48 bits.
`gxHSV32Space`	0x0A03	10 bits per component (hue, saturation, and value); component values range from 0 to 0x3FF. Total storage size for each color value: 32 bits (bits 30–31 are not used).
`gxHLS32Space`	0x0A04	10 bits per component (hue, lightness, and saturation); component values range from 0 to 0x3FF. Total storage size for each color value: 32 bits (bits 30–31 are not used).

Figure 24-31 shows storage formats for the supported HSV color spaces. Formats for the HLS spaces are identical.

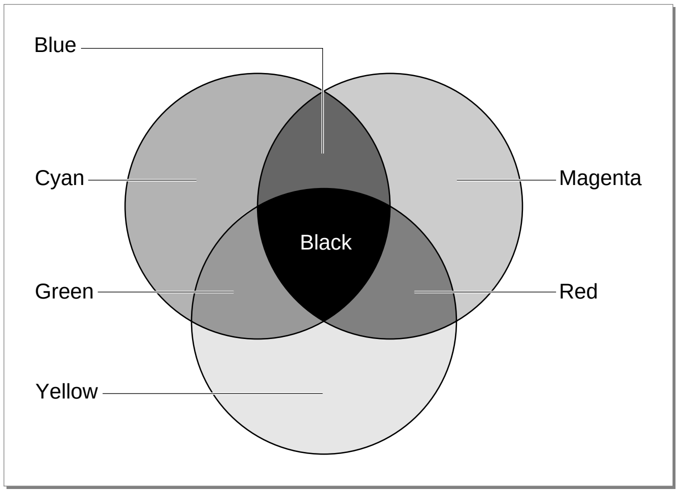
Figure 24-31 Storage formats for HSV color spaces



CMYK Color Spaces

CMYK space is a color space that models the way ink builds up in printing. The name *CMYK* refers to cyan, magenta, yellow, and black. Cyan, magenta, and yellow are the three primary colors in this color space, and red, green, and blue are the three secondaries. Theoretically black is not needed. However, when full-saturation cyan, magenta, and yellow inks are mixed equally on paper, the result is usually a dark brown, rather than black. Therefore, black ink is overprinted in darker areas to give a better appearance. Figure 24-32 shows how the primary colors in CMYK space mix to form other colors.

Figure 24-32 Colors in CMYK color space



Theoretically, the relation between RGB values and CMY values in CMYK space is quite simple:

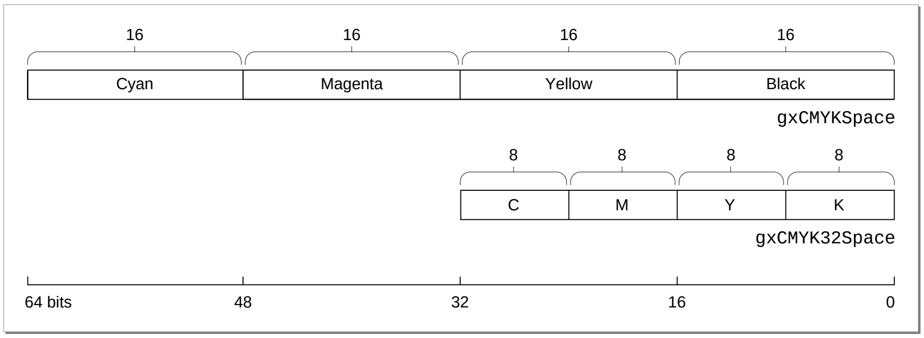
```
Cyan        = 1.0 - red;
Magenta     = 1.0 - green;
Yellow      = 1.0 - blue;
```

(where red, green, and blue intensities are expressed as fractional values varying from 0 to 1). In reality, the process of deriving the cyan, magenta, yellow, and black values from a color expressed in RGB space is complex, involving device-specific, ink-specific, and even paper-specific calculations of the amount of black to add in dark areas (**black generation**), and the amount of other ink to remove (**undercolor removal**) where black is to be printed. The CMYK color spaces supported by QuickDraw GX (and defined in the gxColorSpaces enumeration) are gxCMYKSpace and gxCMYK32Space. See Table 24-6 and Figure 24-33 for details of storage format.

Table 24-6 CMYK color spaces supported by QuickDraw GX

Constant	Enumeration Value	Explanation
gxCMYKSpace	0x0002	16 bits per component (cyan, magenta, yellow, and black); component values range from 0 to 0xFFFF. Total storage size for each color value: 64 bits.
gxCMYK32Space	0x0802	8 bits per component (cyan, magenta, yellow, and black); component values range from 0 to 0xFF. Total storage size for each color value: 64 bits.

Figure 24-33 Storage formats for CMYK color spaces



Universal Color Spaces

Some color spaces allow color to be expressed in a device-independent way. Whereas RGB colors vary with monitor characteristics, and CMYK colors vary with printer and paper characteristics, *universal colors* are meant to be true representations of colors as perceived by the human eye. These color representations, called **universal color spaces,** result from work carried out in

1931 by the Commission Internationale d'Eclairage (CIE), and for that reason are also called *CIE-based color spaces.*

In addition, broadcast-video color space (YIQ) is based on device-independent color characteristics, in that its colors are measured in terms of a standard device. It is therefore considered universal and is discussed in this section.

XYZ Space

There are several CIE-based color spaces, but all are derived from the fundamental **XYZ space.** The XYZ space allows colors to be expressed as a mixture of the three **tristimulus values** X, Y, and Z. The term *tristimulus* comes from the fact that color perception results from the retina of the eye responding to three types of stimuli. After experimentation, the CIE set up a hypothetical set of primaries, XYZ, that correspond to the way the eye's retina behaves.

The CIE defined the primaries so that all visible light maps into a positive mixture of X, Y, and Z, and so that Y correlates approximately to the apparent lightness of a color. Generally, the mixtures of X, Y, and Z components used to describe a color are expressed as percentages ranging from 0% up to, in some cases, just over 100%.

Other universal color spaces based on XYZ space are used primarily to relate some particular aspect of color or some perceptual color difference to XYZ values.

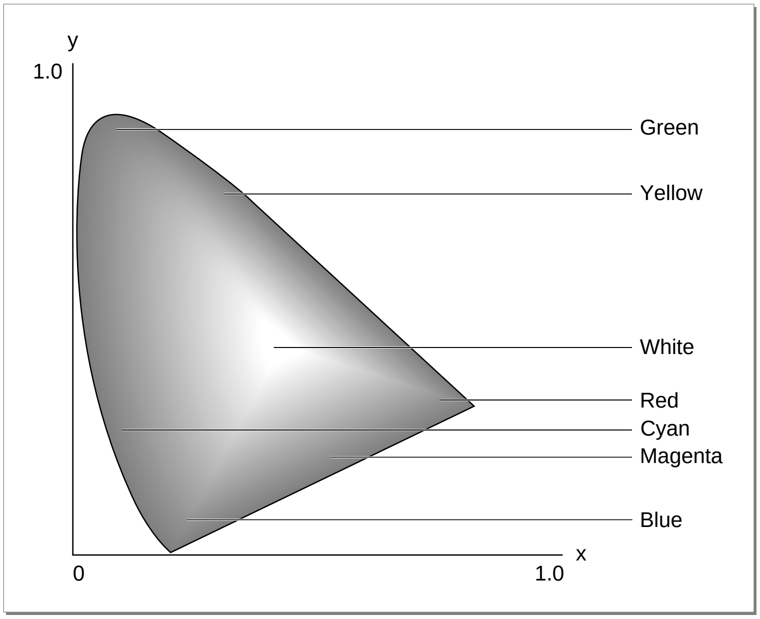
Yxy Space

Yxy space expresses the XYZ values in terms of x and y **chromaticity** coordinates, somewhat analogous to the hue and saturation coordinates of HSV space. The coordinates are shown in the following formulas, used to convert XYZ into Yxy:

```
Y = Y
x = X / (X+Y+Z)
y = Y / (X+Y+Z)
```

Note that the Z tristimulus value is incorporated into the new coordinates, and does not appear by itself. Since Y still correlates to the lightness of a color, the other aspects of the color are found in the chromaticity coordinates x and y. This allows color variation in Yxy space to be plotted on a two-dimensional diagram. Figure 24-34 shows the layout of colors in the x and y plane of Yxy space. Color Plate 4 at the front of this book shows the same plot in color.

Figure 24-34 Yxy chromaticities



L*u*v* Space and L*a*b* Space

One problem with representing colors using the XYZ and Yxy color spaces is that they are perceptually nonlinear: it is not possible to accurately evaluate the perceptual closeness of colors based on their relative positions in XYZ or Yxy space. Colors that are close together in Yxy space may seem very different to observers, and colors that seem very similar to observers may be widely separated in Yxy space.

L*u*v* space is a nonlinear transformation of XYZ space in order to create a perceptually linear color space. **L*a*b* space** is a nonlinear transformation (a third-order approximation) of the Munsell color-notation system (not described here). Both are designed to match perceived color difference with quantitative distance in color space.

Both L*u*v* space and L*a*b* space represent colors relative to a **reference white point,** which is a specific definition of what is considered white light, represented in terms of XYZ space, and usually based on the whitest light that can be generated by a given device. (In that sense L*u*v* and L*a*b* are not completely device independent; two numerically equal colors are truly identical only if they were measured relative to the same white point.)

A primary benefit of using L*u*v* space and L*a*b* space is that the perceived difference between any two colors is proportional to the geometric distance in the color space between their color values. For applications where closeness of color needs to be quantified, such as in colorimetry, gemstone evaluation, or dye matching, use of L*u*v* space or L*a*b* space is common.

The formulas for transforming an XYZ color into an L*u*v* color are

```
if (Y/Yn > 0.008856)
    $L = 116.0 * (Y / Y_n)^{1/3} - 16.0$;
else
    $L = 903.3 * (Y / Y_n)$;
$u = 13.0 * L * (u' - u'_n)$;
$v = 13.0 * L * (v' - v'_n)$;
```

where

```
u' = 4 * x / (X + 15*Y + 3*Z);
v' = 9 * y / (X + 15*Y + 3*Z);
```

and u'_n, v'_n, and Y_n are the u', v', and Y values for the reference white point.

Similarly, the formulas for transforming an XYZ color into an L*a*b* color are

```
if (Y/Yn > 0.008856)
    $L = 116.0 * (Y / Y_n)^{1/3} - 16.0$;
else
    $L = 903.3 * (Y / Y_n)$
$a = 500.0 * ( (X / X_n)^{1/3} - (Y / Y_n)^{1/3} )$;
$b = 500.0 * ( (Y / Y_n)^{1/3} - (Z / Z_n)^{1/3} )$;
```

where X_n, Y_n, and Z_n are the XYZ values for the reference white point.

Formats for XYZ-Based Color Spaces

The universal color spaces supported by QuickDraw GX (and defined in the gxColorSpaces enumeration) are gxYXYSpace, gxXYZSpace, gxLUVSpace, gxLABSpace, gxYXY32Space, gxXYZ32Space, gxLUV32Space, and gxLAB32Space. See Table 24-7 and

Figure 24-35 for details of storage format. Note that the ranges of values for the components differ significantly among the different color spaces.

Table 24-7 Universal color spaces supported by QuickDraw GX

Constant	Enumeration Value	Explanation
`gxYXYSpace`	0x0005	16 bits per component (Y, x, and y); component values range from 0 (0%) to 0xFFFF (100%). Total storage size for each color value: 48 bits.
`gxXYZSpace`	0x0006	16 bits per component (X, Y, and Z). Component values range from 0 (0%) to 0xFFFF (200%; a value of 0x8000 represents 100%). Total storage size for each color value: 48 bits.
`gxLUVSpace`	0x0007	16 bits per component (L*, u*, and v*). The L* component values range from 0 (0%) to 0xFFFF (100% of white-point luminance). The u* and v* component values range from 0 (–1) to 0xFFFF (+1). Total storage size for each color value: 48 bits.
`gxLABSpace`	0x0008	16 bits per component (L*, a*, and b*). The L* component values range from 0 (0%) to 0xFFFF (100% of white-point luminance). The a* and b* component values range from 0 (–1) to 0xFFFF (+1). Total storage size for each color value: 48 bits.
`gxYXY32Space`	0x0A05	10 bits per component (Y, x, and y); component values range from 0 (0%) to 0x3FF (100%). Total storage size for each color value: 32 bits (bits 30 and 31 not used).
`gxXYZ32Space`	0x0A06	10 bits per component (X, Y, and Z). Component values range from 0 (0%) to 0x3FF (200%; a value of 0x200 represents 100%). Total storage size for each color value: 32 bits (bits 30 and 31 not used).

Table 24-7 Universal color spaces supported by QuickDraw GX (continued)

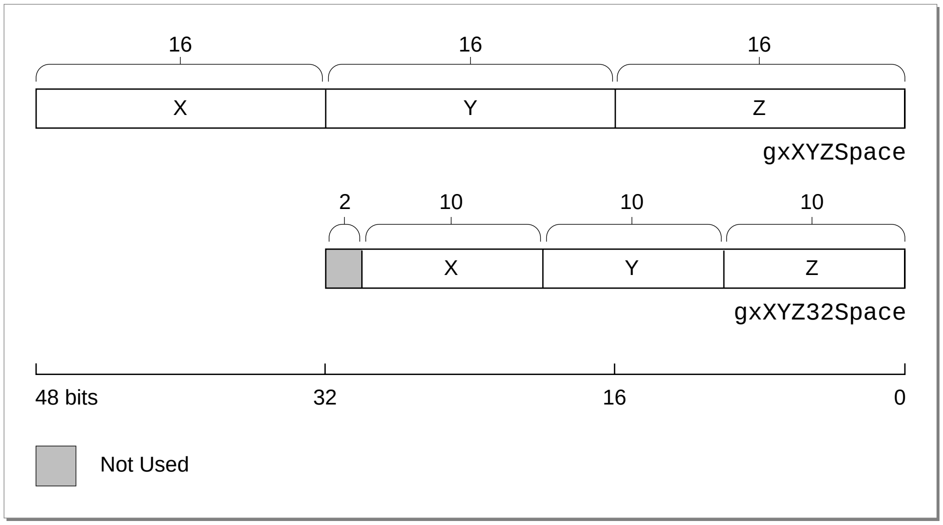
Constant	Enumeration Value	Explanation
`gxLUV32Space`	0x0A07	10 bits per component (L*, u*, and v*). The L* component values range from 0 (0%) to 0x3FF (100% of white-point luminance). The u* and v* component values range from 0 (–1) to 0x3FF (+1). Total storage size for each color value: 32 bits (bits 30 and 31 not used).
`gxLAB32Space`	0x0A08	10 bits per component (L*, a*, and b*). The L* component values range from 0 (0%) to 0x3FF (100% of white-point luminance). The a* and b* component values range from 0 (–1) to 0x3FF (+1). Total storage size for each color value: 32 bits (bits 30 and 31 not used).

NOTE Because u*, v*, a*, and b* are normally signed numbers between 1.0 and -1.0, you can convert them into short integers as follows:

```
anUnsignedshort = ((aFloat + 1.0)/2) * 65535.0;
```

Figure 24-35 shows storage formats for the supported XYZ color spaces. Formats for the Yxy, L*u*v*, and L*a*b* spaces are identical.

Figure 24-35 Storage formats for XYZ color spaces



Video Color Spaces

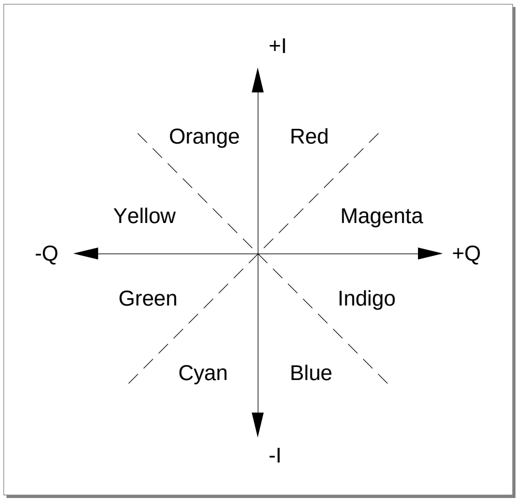
YIQ space is sometimes called *video color space*. It is based on the way a specific kind of RGB data is broken down for color television transmission. The three dimensions that describe these color spaces are Y, I, and Q, in which Y represents luminance and the other two components carry color information.

Because the Y channel represents luminance it can be used alone; the Y channel is the only channel used in black and white television. The I and Q channels are called *color difference* channels: the Y channel is split between them. The notations “I” and “Q” stand for “in phase” and “in quadrature,” respectively, referring to the method by which all of the channels are combined into a signal for broadcast.

QuickDraw GX also defines NTSC and PAL color spaces. NTSC space corresponds to the color encoding used for color broadcasting in the United States, whereas PAL space corresponds to the color encoding used in Europe. NTSC and PAL have different screen resolutions, frequencies, and are otherwise incompatible, but in terms of how color values are calculated, NTSC space and PAL space are both identical to YIQ space.

In YIQ space, the Y component can vary from 0 (black) to its maximum value (full luminance). I and Q are normally signed values, so they are centered around 0. Figure 24-36 illustrates how colors map into the I and Q dimensions of YIQ space.

Figure 24-36 The I and Q axes in YIQ color space



The video color spaces supported by QuickDraw GX (and defined in the `gxColorSpaces` enumeration) are `gxYIQSpace`, `gxNTSCSpace`, `gxPALSpace`, `gxYIQ32Space`, `gxNTSC32Space`, and `gxPAL32Space`. See Table 24-8 and

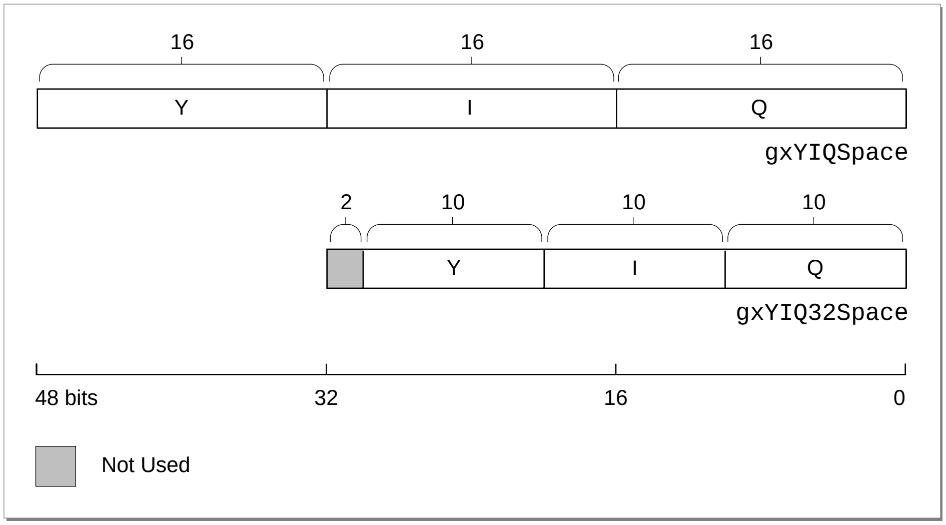
Figure 24-37 for details of storage format. In each of these spaces, a color value is represented by Y, I, and Q color components.

Table 24-8 Video color spaces supported by QuickDraw GX

Constant	Enumeration Value	Explanation
`gxYIQSpace`	0x0009	16 bits per component (Y, I, and Q); Y-component values range from 0 to 0xFFFF; I- and Q-component values range from –0x7FFF to +0x7FFF. Total storage size for each color value: 48 bits.
`gxNTSCSpace`	0x0009	(same as `gxYIQSpace`)
`gxPALSpace`	0x0009	(same as `gxYIQSpace`)
`gxYIQ32Space`	0x0A09	10 bits per component (Y, I, and Q); Y-component values range from 0 to 0x3FF; I- and Q-component values range from –0x1FF to +0x1FF. Total storage size for each color value: 32 bits (bits 30 and 31 are not used).
`gxNTSC32Space`	0x0A09	(same as `gxYIQ32Space`)
`gxPAL32Space`	0x0A09	(same as `gxYIQ32Space`)

Figure 24-37 shows storage formats for the supported YIQ color spaces. Formats for the NTSC and PAL spaces are identical.

Figure 24-37 Storage formats for YIQ color spaces



You can find more information on the theories of color and the various color spaces in the following publications:

Measuring Color, by R.W.G. Hunt, John Wiley & Sons, New York, 1987.

Illumination and Color in Computer Generated Imagery, by Roy Hall, Springer-Verlag, New York, 1989.

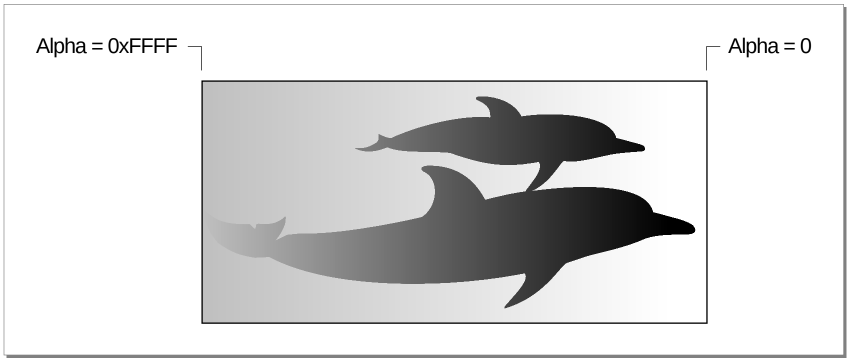
Color Spaces With Alpha Channels

QuickDraw GX supports the use of an alpha channel in one luminance-based color space (`gxGrayASpace`) and two RGB color spaces (`gxRGBASpace` and `gxARGB32Space`). An **alpha channel** is a component in a color space whose value typically determines the opacity of the color expressed by the other components. An alpha-channel value of 0 in a color means that the color is completely transparent, and a maximum value means that the color is completely opaque. A value in between means that the color is partially transparent.

How transparency is handled in drawing depends on the transfer mode used when the color is drawn. (Transfer modes are discussed in “Transfer Modes” (page 814).) Typically, however, transparency in a color being drawn—the source color— means that the existing color at the location where drawing occurs—the destination color—shows through. Where the source is completely opaque, the destination is completely covered and is invisible; where the source is completely transparent, the destination shows through unchanged and the source is invisible.

Figure 24-38 shows an example in which a uniform gray image (in `gxGrayASpace`) is drawn over a black-and-white image. The gray color of the source is uniform across the rectangle, but the alpha-channel value decreases from 0xFFFF on the left to 0 on the right. As the alpha value decreases rightward, more and more of the destination color shows through.

Figure 24-38 Showing color transparency with an alpha channel



Color-Component Values, Color Values, and Colors

Each of the color spaces described in this chapter requires one or more numeric values in a particular format to specify a color. This section describes the data types and structures with which QuickDraw GX describes colors in its color spaces.

Each dimension, or component, in a color space has a **color-component value.** In the fundamental, unpacked QuickDraw GX color spaces—those with 16 bits per component—each color-component value is of type `gxColorValue`:

```
typedef unsigned short gxColorValue;
```

A color-component value can vary from 0 to 65,535 (0xFFFF), although the numerical interpretation of that range is different for different color spaces. In most cases, color-component intensities are interpreted numerically as varying between 0 and 1.0; for that reason, QuickDraw GX provides the constant `gxColorValue1` to represent 0xFFFF.

Depending on the color space used, one, two, three, or four color-component values combine to make a **color value.** A color value is a structure; it is the complete specification of a color in a given color space. QuickDraw GX supports 13 color-value formats, representing the fundamental 16-bits-per-component color spaces; all color operations in memory use one of those formats. The color-value formats are described in the section "Color-Component Values, Color Values, and Colors" (page 812). For example, an RGB color value has this format:

```
struct gxRGBColor{
    gxColorValue        red;
    gxColorValue        green;
    gxColorValue        blue;
};
```

This is exactly the storage format for colors in `gxRGBSpace`. However, colors stored in `gxRGB16Space` or `gxRGB32Space` have a packed storage format, and need to be converted to `gxRGBColor` format when they are used. QuickDraw GX takes care of this for you; as far as your application is concerned, you can always manipulate colors in the color space you have specified.

A color value plus a specification of the color space it belongs to constitute a **color** in QuickDraw GX. A color is defined by the `gxColor` structure:

```
struct gxColor{
    gxColorSpace                space;
    gxColorProfile              profile;
    union {
            struct gxCMYKColor              cmyk;
            struct gxRGBColor               rgb;
            struct gxRGBAColor              rgba;
            struct gxHSVColor               hsv;
            struct gxHLSColor               hls;
            struct gxXYZColor               xyz;
            struct gxYXYColor               yxy;
            struct gxLUVColor               luv;
            struct gxLABColor               lab;
            struct gxYIQColor               yiq;
            gxColorValue                    gray;
            struct gxGrayAColor             graya;
            unsigned short                  pixel16;
            unsigned long                   pixel32;
            struct gxIndexedColor           indexed;
            gxColorValue                    component[4];
    } element;
};
```

Each `gxColor` structure holds the specification of a single color. Note that, besides the basic color-value formats such as `gxRGBColor` and `gxXYZColor`, a QuickDraw GX color can contain a 16-bit or 32-bit pixel value, and you can also access the color as an array of color-component values. Each of the color values in the `element` union of the `gxColor` structure is described in the section "Color Structure" (page 882).

Transfer Modes

Basically, ink objects exist to specify two important characteristics of a shape: its color and the transfer mode to draw it with. Transfer modes are described here.

Transfer modes specify how a shape's color is transferred onto a device. The color of a shape to be drawn (the **source color**) interacts with the existing color (the **destination color**) on the device it is drawn to. The color that results from that interaction is called the **result color.** The result color is the color of the destination after the drawing occurs.

Note that colors from different color spaces can be used. The source and destination colors are converted to the transfer mode's color space, and the resulting color is then reconverted to the destination color space.

Transfer Mode Types

Transfer modes can be specified by type, also called **component mode.** Transfer mode types in QuickDraw GX are called component modes because QuickDraw GX allows each color component to have its own transfer mode type. In RGB color space, for example, the red component of the color may be drawn with a different transfer mode type than the blue component.

QuickDraw GX supports several conceptual categories of component modes:

- arithmetic
- Boolean
- pseudo-Boolean
- highlight
- alpha-channel

The characteristics of and most typical uses for the component modes within each category are summarized in the following subsections.

Note
Even though QuickDraw GX supports 18 different component modes, most applications in most situations need only one, an arithmetic mode called **copy mode.** In copy mode, the source color completely replaces the destination color. Copy mode is the default transfer mode in QuickDraw GX; therefore, you need information about other transfer modes only if you want them for special effects. ◆

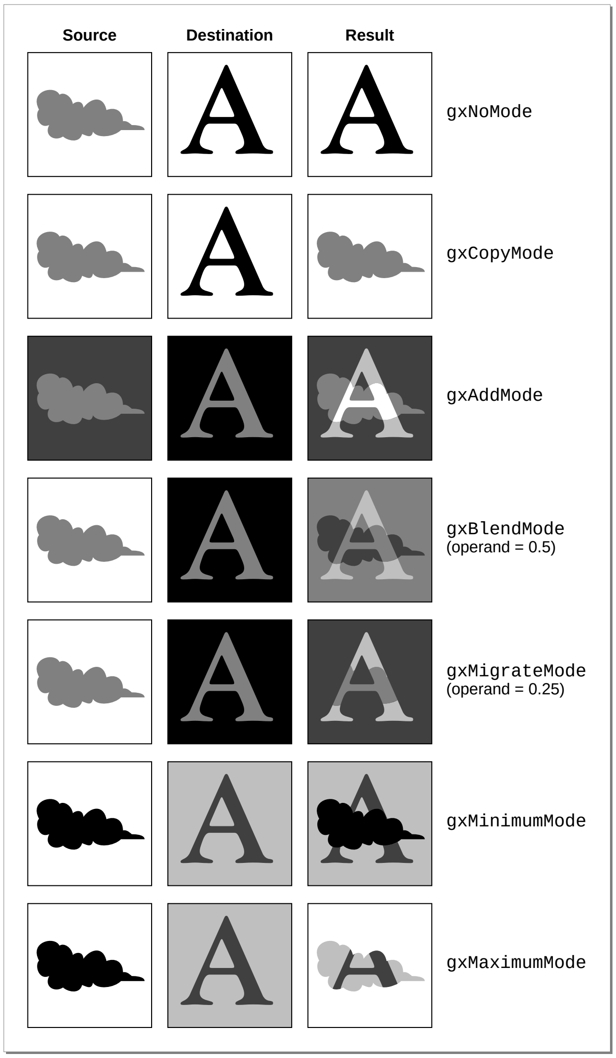
Arithmetic Transfer Modes

In arithmetic transfer modes, the numerical values of source and destination for a color component are combined arithmetically to determine the result value for that color component. In most color spaces, a color component value can vary from 0 (no intensity) to 0xFFFF (maximum intensity). You can also use the constant `gxColorValue1` to represent maximum intensity (0xFFFF).

Figure 24-39 shows examples of drawing with the arithmetic transfer modes. In each case, the source image (left) combines with the destination image (center) to produce the result image (right). You can think of the images either as two bitmaps, or as two source shapes (cloud and background) that are drawn over two destination shapes (letter and background).

Each example shows how transfer mode affects drawing within a single color component (reflected as shades of gray in the figure, where black equals 0 and white equals 0xFFFF). The constant that specifies the transfer mode type is shown to the right of each example. Note also that two of the arithmetic transfer modes use an **operand,** a numerical value that affects the outcome of the transfer-mode operation.

Figure 24-39 Arithmetic transfer modes



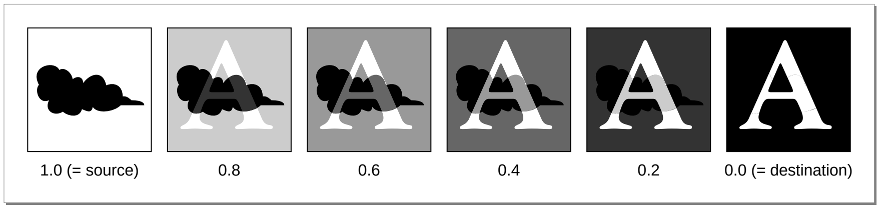
The constants that define transfer mode type are defined in the `gxComponentModes` enumeration. The arithmetic modes have the following values and meanings:

Constant	Value	Explanation
`gxNoMode`	0	No mode. No transfer occurs. For this component of the color, the destination is left as it was. This mode is useful for suppressing drawing when certain logical conditions are met, or for not drawing one color component while allowing other components to be drawn.
`gxCopyMode`	1	Copy mode. The source color component is copied to the destination. The destination component is ignored. This is the most common transfer mode, and is the default for QuickDraw GX.
`gxAddMode`	2	Add mode. The source color component is added to the destination component, but the result is not allowed to exceed the maximum value (`0xFFFF` or `gxColorValue1`; white in Figure 24-39).
`gxBlendMode`	3	Blend mode. The result is the average of the source and destination color components, weighted by a ratio specified by the operand component (0.5 in Figure 24-39). The operand varies from 0 (all destination) to `0xFFFF` or `gxColorValue1` (all source), although it is customary to interpret it as varying between `0` and `1`.

Constant	Value	Explanation
`gxMigrateMode`	4	Migrate mode. The destination color component is moved toward the source component by the value of the step specified in the operand component (0.25, or `0x4000` in Figure 24-39). Migrate mode is similar to blend mode, except that the change in destination component is an absolute amount, rather than a proportion of the difference between it and the source component. If the source has a greater color component value than the destination, the migration is positive; if the destination has a greater value than the source, the migration is negative. In either case, the amount of migration cannot be greater than the difference between the destination and the source values.
`gxMinimumMode`	5	Minimum mode. The source component replaces the destination component only if the source component has a smaller value. (In Figure 24-39, drawing occurs only within the area occupied by the cloud.)
`gxMaximumMode`	6	Maximum mode. The source component replaces the destination component only if the source component has a larger value. (In Figure 24-39, drawing occurs only outside of the area occupied by the cloud.)

The `operand` parameter is used by blend mode to specify the ratio of source and destination component. It is used by migrate mode to specify the step size by which the destination component moves toward the source component. Figure 24-40 shows examples of the result of drawing with blend mode, using several different values for the operand.

Figure 24-40 Blend example with different operand values



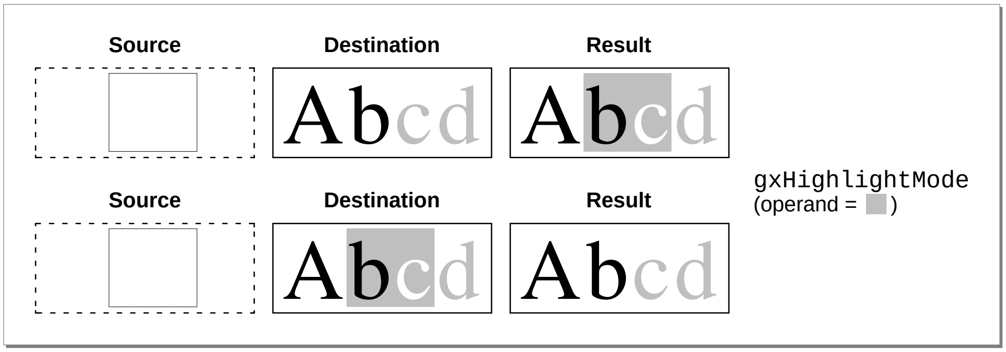
Highlight Transfer Mode

The highlight transfer mode is used for highlighting in color applications. It is most commonly used to draw (and clear) a colored rectangle around a selection, without altering the color of the item or items selected. In text, it gives the effect of drawing over the letters with a highlighting pen.

Like some of the arithmetic transfer modes, highlight mode uses an operand to control the outcome of the highlighting operation. Highlight mode operates by replacing the source color with the operand color, and the operand color with the source color, in the destination.

The upper row of images in Figure 24-41 shows a simple example of the application of highlight mode. The operand value is represented with shading rather than as a number, to illustrate how its color affects colors in the image. The source shape is a white rectangle that is drawn over the two middle letters in the destination image. (The gray letters in the line of text in the destination image represent the same color-component value as the operand, and the white area around the letters in the destination represents the same color-component value as the source.)

Figure 24-41 Highlight transfer mode



Note that black in the destination is unaffected, whereas white becomes gray and gray becomes white. A single constant specifies highlight mode, with the following value and meaning:

Constant	Value	Explanation
`gxHighlightMode`	7	Highlight mode. The source component and operand component are swapped in the destination. Other components in the destination are ignored.

In highlight mode, the source color can be thought of as the “background” color that is to be highlighted, and the operand color is the color of the highlighting pen. As the lower set of images in Figure 24-41 shows, redrawing a highlighted selection causes the source and operand colors to swap once more, effectively removing the highlighting.

The operand for highlight mode is a normal color component value that varies from 0 (no intensity) to the maximum intensity permitted for that component (normally `0xFFFF`, or `gxColorValue1`).

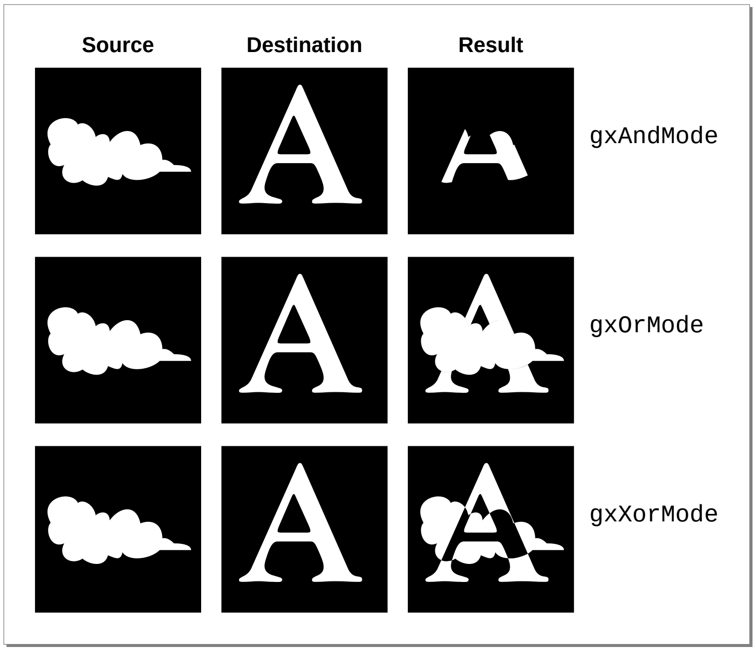
QuickDraw GX applies highlight mode only if all components in the color space specify it. An error occurs if some components specify highlight mode and others do not.

Boolean Transfer Modes

In Boolean transfer modes, the result value for a color component is determined by bit operations performed on the source and destination component values. Boolean transfer modes are most common in black-and-white drawing; in any bit depth other than 1, they yield results that can be difficult to predict because they depend on the states of the individual bits in each color-component value.

Figure 24-42 shows examples of drawing with the Boolean transfer modes at a bit depth of 1. In each case, the source image combines with the destination image to produce the result image. In these examples, black represents a bit value of 0 (clear), and white represents a bit value of 1 (set). The constant that specifies the transfer mode type is shown to the right of each example.

Figure 24-42 Boolean transfer modes (1-bit depth)



The Boolean modes have the following values and meanings:

Constant	Value	Explanation
`gxAndMode`	8	AND mode. The bits of the source color and destination color are combined using an AND operation. Only bits that are set in both source and destination remain set in the result.
`gxOrMode`	9	OR mode. The bits of the source color and destination color are combined using an OR operation. Bits that are set in either the source or the destination or in both are set in the result.
`gxXorMode`	10	XOR mode. The bits of the source color and destination color are combined using an exclusive-OR (XOR) operation. Bits that are set in the source but not the destination, and bits that are set in the destination but not the source, are set in the result. All other bits are cleared in the result.

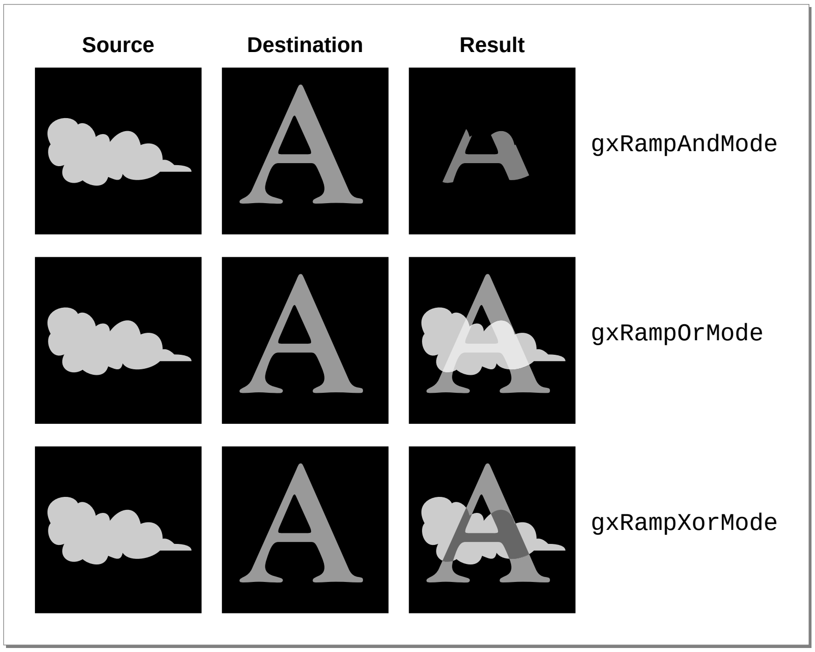
Even though they are most easily explained in terms of single-bit depths, Boolean modes are not restricted to 1-bit drawing. They can be used with any kind of color values.

Pseudo-Boolean Transfer Modes

In pseudo-Boolean transfer modes, the result value for a color component is determined by normalizing the source and destination values and performing a simple arithmetic operation, to achieve consistent and predictable results analogous to 1-bit Boolean operations.

Figure 24-43 shows examples of drawing with the pseudo-Boolean transfer modes. In each case, the source image combines with the destination image to produce the result image. The constant that specifies the transfer mode type is shown to the right of each example.

Figure 24-43 Pseudo-Boolean transfer modes



The constants for the pseudo-Boolean component modes have the following values and meanings:

Constant	Value	Explanation
`gxRampAndMode`	11	Ramp-AND mode. The source and destination color components are treated as ranging from 0 to 1; their product (source × destination) is returned.
`gxRampOrMode`	12	Ramp-OR mode. The source and destination color components are treated as ranging from 0 to 1; the result of (source + destination – source × destination) is returned.
`gxRampXorMode`	13	Ramp-XOR mode. The source and destination color components are treated as ranging from 0 to 1; the result of (source + destination – 2 × source × destination) is returned.

Note that the pseudo-Boolean and Boolean modes are similar in several ways:

- The mode gxRampAndMode is similar to gxAndMode in that nonzero values occur in the result only where both source and destination are nonzero.
- The mode gxRampOrMode is similar to gxOrMode in that nonzero values occur in the result wherever either the source or the destination is nonzero.
- The mode gxRampXorMode is similar to gxXorMode in that the result is close to zero wherever the source and destination are close to each other in value.

The difference between the pseudo-Boolean and Boolean modes is that, for multi-bit pixel depths, the results for gxRampAndMode, gxRampOrMode, and gxRampXorMode are predictable and vary smoothly and continuously with component intensity. For 1-bit depths, these modes are identical to their Boolean equivalents.

The pseudo-Boolean modes are commonly used as component modes for alpha channels in color spaces that have an alpha channel. For more information, see “Alpha-Channel Transfer Modes” (page 824).

Alpha-Channel Transfer Modes

Several QuickDraw GX color spaces (gxRGBASpace, gxARGB32Space and gxGrayASpace) have an **alpha channel.** This is an additional color component that controls the opacity or transparency of a color. For example, a red pixel in a source image can be completely opaque, in which case it typically retains its red color when drawn over a blue pixel in the destination image. Or, the pixel can be completely transparent, in which case it typically loses all its color and turns totally blue when drawn over a blue pixel. Or, it can have an opacity of, say, 0x7FFF (50%), in which case it typically turns magenta when drawn over a blue pixel.

Alpha channel values can be used to allow parts of one image to show through “holes” in another, to show translucency in objects that are drawn over other objects, and to perform anti-aliasing (smoothing of jagged edges) by giving feathered, semi-transparent borders to opaque objects.

When assigning transfer modes to colors with an alpha channel, you typically use two different kinds of modes:

- To get the proper result color for each color component, you use an alpha-channel transfer mode. These modes take alpha-channel values into account when calculating result values for the color components.

- To get the proper result opacity for the alpha channel itself, you typically use an arithmetic or pseudo-Boolean transfer mode.

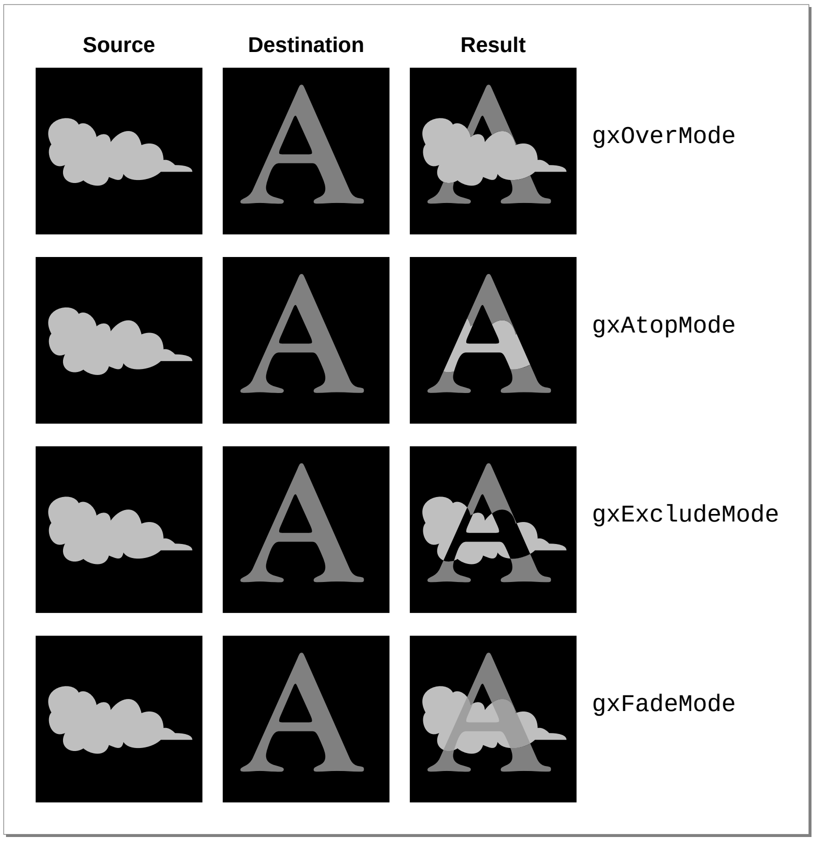
This section describes how the different modes within each category work to give you the results you want.

Modes for the Color Components

Figure 24-44 shows examples of how values for a color component might be calculated, given a source image and a destination image consisting of objects (or pixels) that differ in opacity. In each example, the source image (an opaque, light gray cloud against a transparent black background) combines with the destination image (an opaque, dark gray "A" on a transparent black background), to form the result image. The constant that specifies the transfer mode type is shown to the right of each example.

Each example shows how the alpha-channel transfer mode affects drawing within a single color component. The mode takes into account not only the source and destination color components, but the source and destination opacities as well.

Figure 24-44 Alpha-channel transfer modes



The constants for the alpha-channel component modes have the following values and meanings:

Constant	Value	Explanation
`gxOverMode`	14	Over mode. The source color is copied to the destination, and the source transparency controls where the destination color shows through. Where both are transparent, no drawing occurs (result equals destination).

Constant	Value	Explanation
`gxAtopMode`	15	Atop mode. The source color is placed over the destination, but the resulting destination retains the original destination's transparency. The effect is that opaque parts of the source are clipped to cover only opaque parts of the destination.
`gxExcludeMode`	16	Exclude mode. The destination color remains visible only where the source is transparent, and the source color is copied anywhere the destination is transparent. Where both are transparent, no drawing occurs (result equals destination); where both are opaque, the result color is 0 (no intensity).
`gxFadeMode`	17	Fade mode. The source is blended with the destination, using the relative alpha values as the ratio for the blend. Where both are transparent, the result is the average of the source and the destination).

As Figure 24-44 shows, the `gxOverMode` mode is similar to the arithmetic transfer mode `gxCopyMode`, except that it allows for transparency in the source image. Likewise, the `gxAtopMode` mode is similar to `gxCopyMode`, but it preserves the transparency of the destination image by clipping the opaque source to the destination image. The `gxExcludeMode` mode is somewhat like the Boolean transfer mode `gxXorMode`, in that opaque parts of each image appear only where the other is not opaque. The `gxFadeMode` mode is like the arithmetic `gxBlendMode`, except that the operand that controls the blend ratio is determined by the relative opacities.

Note that the images shown in Figure 24-44 are very simple and their opacities are either 0 (completely transparent) or `gxColorValue1` (completely opaque). Because an alpha component can have a wide range of partial opacities, very sophisticated translucency and color-blending effects are possible, as well as the simple masking effects shown here.

The exact formulas for determining result color are the following. In these formulas, `sA` = source alpha-channel value; `dA` = destination alpha-channel value;
`sC` = source color-component value; `dC` = destination color-component value; and `rC` = result color-component value.

- For `gxOverMode`:

```
rC = (sA x (sC - dA x dC) + dA x dC) / (sA + dA - sA x dA)
```

- For `gxAtopMode`:

```
rC = dC + sA x (sC - dC)
```

- For `gxExcludeMode`:

```
rC =    (sA x sC + dA x dC- sA x dA x (sC + dC)) /
        (sA + dA - sA x dA)
```

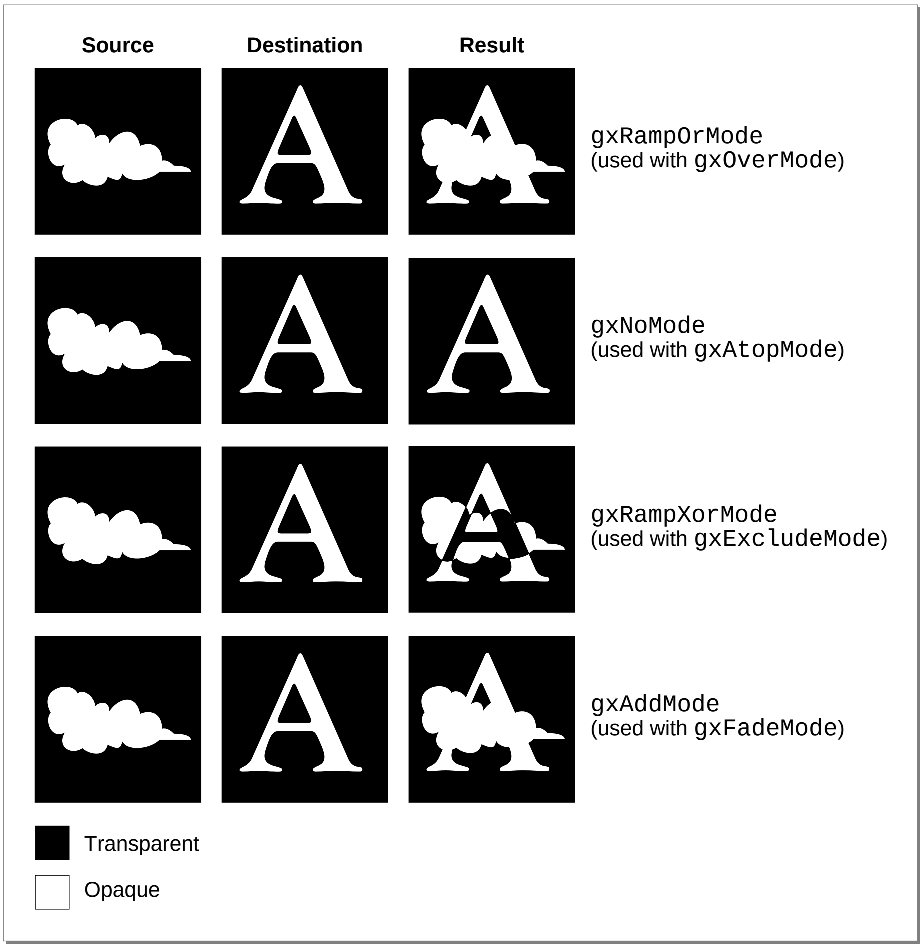
- For `gxFadeMode`:

```
rC = sA x sC + dA x dC / (sA + dA)
```

Modes for the Alpha Channel

For calculating the result value for the alpha channel itself, you typically use one of the arithmetic or pseudo-Boolean transfer modes presented in the previous sections—one that takes into account only the source and destination opacities. Figure 24-45 shows typical modes used and their effects on opacity, using the same images is those presented in Figure 24-44. In this figure, black represents complete transparency and white represents complete opacity. (If alpha-channel values between the two extremes existed in these examples, they would be shown in shades of gray.)

Figure 24-45 Typical modes used to determine result opacity for the alpha channel



Note from Figure 24-45 that the mode you use to determine the result opacity of the alpha channel usually depends on what alpha-channel mode you use to get color-component values:

- Use `gxRampOrMode` to calculate result alpha-channel values if you want the opacities of both source and destination summed proportionally (in a pseudo-Boolean manner; see the description of `gxRampOrMode` to achieve a result opacity. Thus, if you use `gxOverMode` for the color-components, you would typically use `gxRampOrMode` for the alpha channel.

- Use `gxNoMode` to calculate result alpha-channel values if you want the opacity of the destination to remain unchanged. Thus, if you use `gxAtopMode` for the color-components, you would typically use `gxNoMode` for the alpha channel.
- Use `gxRampXorMode` to calculate result alpha-channel values if you want a maximum result opacity where there is a maximum difference in opacities between source and destination. Thus, if you use `gxExcludeMode` for the color-components, you would typically use `gxRampXorMode` for the alpha channel.
- Use `gxAddMode` to calculate result alpha-channel values if you want the result opacity to reflect the sum of the opacities of the source and destination (pinned to the maximum permitted value). Thus, if you use `gxFadeMode` for the color-components, you would typically use `gxAddMode` for the alpha channel.

Note
When converting a color from a color space that does not have an alpha channel to one that does, QuickDraw GX sets the alpha channel intensity to maximum (opaque). When a color is converted from a color space that does have an alpha channel to one that does not, the alpha channel is lost. ◆

Transparency Ramps and Anti-Aliasing

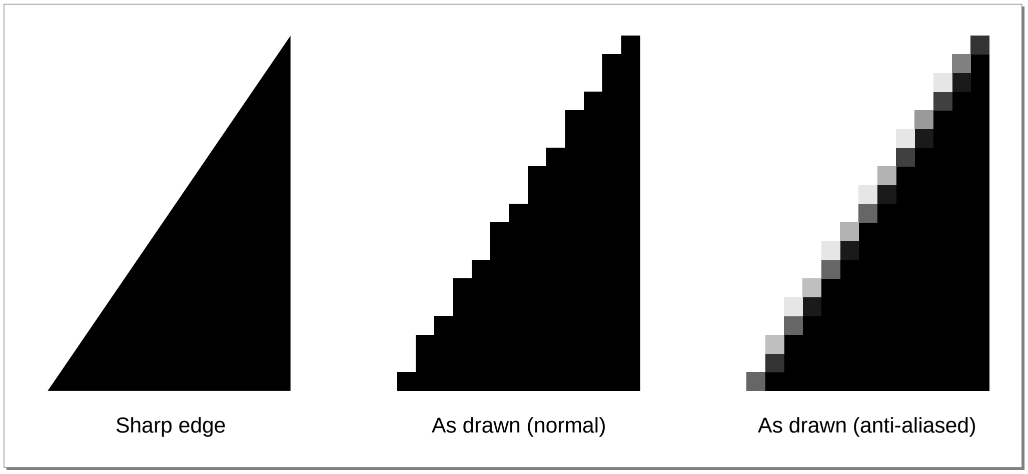
Two common applications for alpha-channel colors involve making objects or images partially opaque to give a translucent effect, and smoothing jagged edges on objects drawn at low resolution.

You can create a bitmap in which the alpha-channel values of the pixels vary smoothly in one or more directions, thus creating a transparency ramp that allows the destination image to show through the source image to varying degrees across the bitmap. Color Plate 2 at the front of this book, for example, shows the kind of effect that can be achieved with a simple alpha-channel ramp.

The smoothing of jagged edges on displayed objects is called **anti-aliasing.** You can perform anti-aliasing by modifying the alpha-channel values of the pixels surrounding the edges of an opaque object. You make an individual pixel more or less opaque, based on the proportion of that pixel that the object is computed to cover.

In Figure 24-46, for example, the left image shows the computed position of the edge of a shape in a bitmap. The center image shows how that edge is displayed normally, given the resolution of the bitmap. The right image shows that edge as it might be displayed with anti-aliasing applied. The apparent jaggedness is decreased because pixels near the edge allow the background to show through to varying degrees.

Figure 24-46 Anti-aliasing



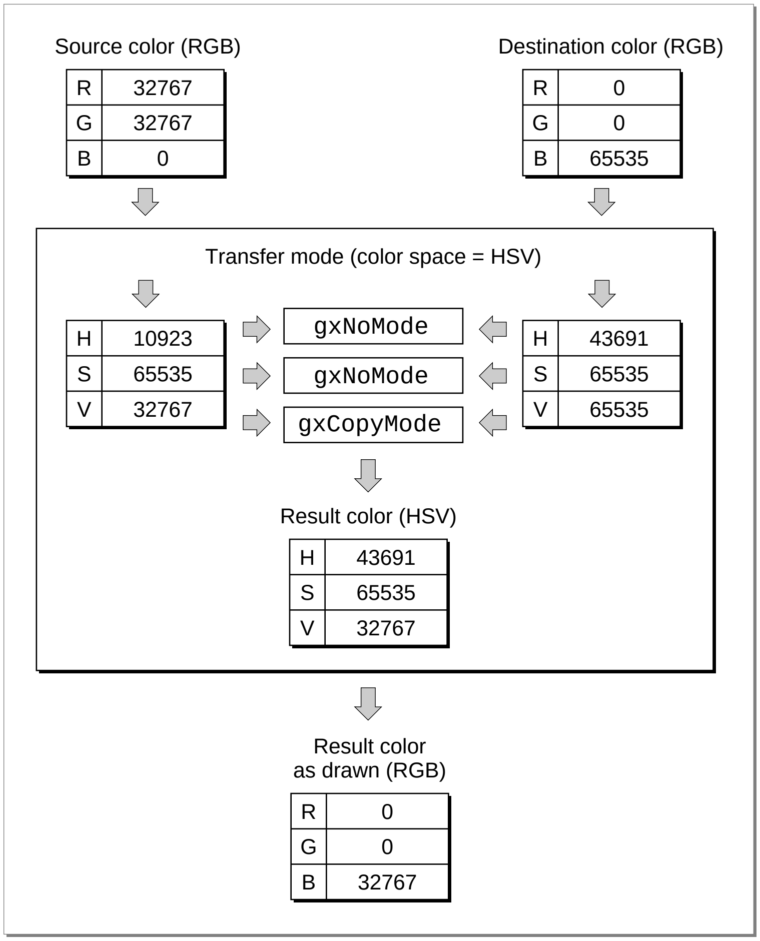
Transfer Mode Color Space

The `space` field in the transfer mode structure specifies the color space the transfer mode calculations take place in. This space does not need to be the same as the color space specified by the ink's color, or the destination color space as specified by the view port or view device with which the ink is associated. The source and destination colors are converted into the color space that provides the context for the transfer, and the resulting color is then reconverted to the destination color space. Keep in mind that all transfer mode computations take place in the transfer mode's color space.

You needn't convert color values among different spaces yourself in order to use a different transfer mode. The transfer mode operation automatically converts colors from the ink's color space and the view device's color space, manipulates them, and then converts the result color back to the view device's color space for drawing. In creating shapes, you can work in whatever color space is convenient for you; when drawing, you can use any transfer mode color space you want; and neither color space need be the same as the color space used by the view device to which you are drawing.

Figure 24-47, for example, shows a source color in RGB space as specified in an ink object, a destination color in RGB space as specified by a monitor, and a transfer mode color space of HSV, as specified by the application. The component modes selected mean that the hue and saturation of the destination are preserved, but the value (lightness) of the source is maintained. QuickDraw GX automatically performs all necessary conversions.

Figure 24-47 Automatic conversion of color values during a transfer mode operation



The transfer mode color space defines how many components are required to perform the transfer mode operation. Monochromatic (grayscale) color spaces require only one component to be filled out. Alpha-channel spaces and CMYK space require four components to be filled out. All other spaces require three components to be filled out.

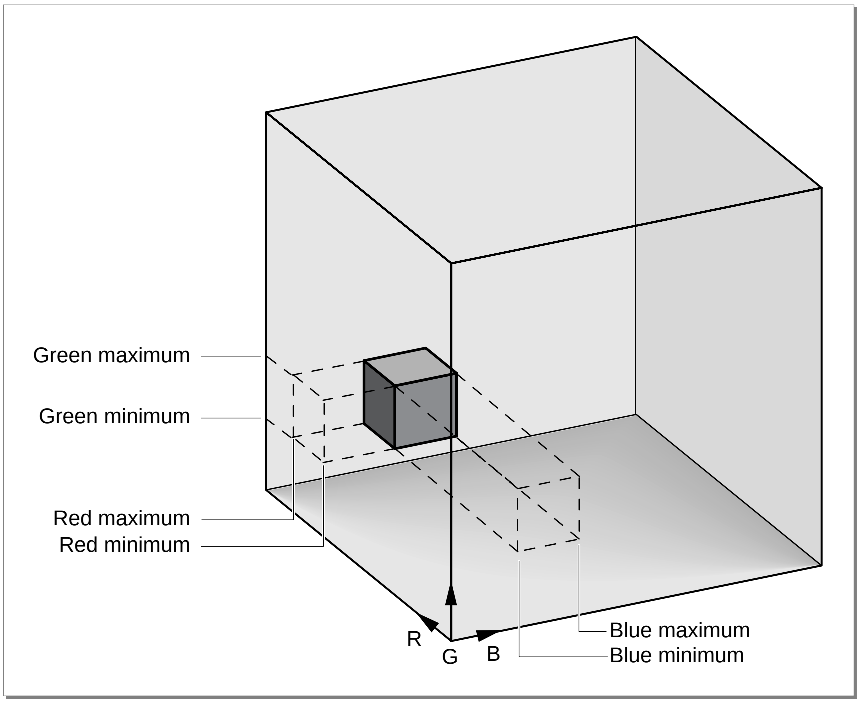
Note
Choosing a different color space can radically affect the behavior of a transfer mode. For example, if your transfer mode uses RGB space, and you have specified `gxCopyMode` for the component mode of `component[0]` and `gxNoMode` for the other components in the transfer mode structure, drawing will transfer only the red component of your source image to the destination and leave the blue and green components of the destination as they are. If you then change the transfer mode color space to HSV and redraw, all hues in your source image will be transferred to the destination, but with the brightnesses and saturations of the original destination image. ◆

Color Limits

Transfer mode operations allow you to specify limits on the acceptable input values for the source or destination color, and on the acceptable output values for the result color. For example, in converting CMYK color to RGB, you may wish to limit the intensities to values that can be displayed without oversaturating the phosphors on a monitor's screen. Or, to create a special effect, you may want to draw only the extreme light and dark portions of an image, leaving out its midrange entirely.

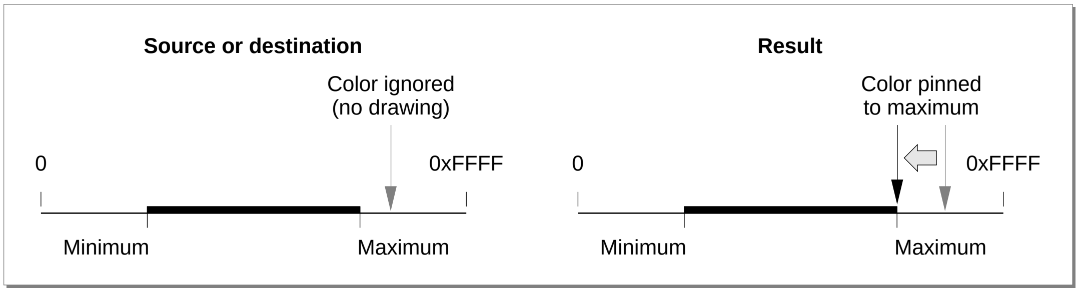
Each color component in the `component` field of the transfer mode structure can have a maximum and a minimum permitted value. The permissible ranges can be interpreted as shown in Figure 24-48. In the figure, the large cube represents all of RGB space; the small cube represents one possible example of the limits that could be imposed on allowable values for all three components.

Figure 24-48 Maximum and minimum color-component values in RGB space



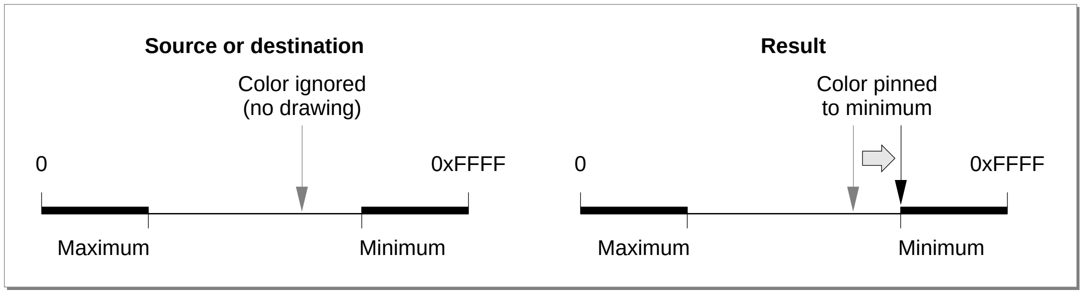
In the case of source and destination colors, color values outside the range of acceptable values (that is, outside the small cube in Figure 24-48) are ignored; if any single component value is outside of its acceptable range, no drawing occurs at all for that color. In the case of the calculated colors that result from a given transfer mode operation, color values outside of the acceptable range are **pinned** to, or moved so that they don't exceed, the nearest acceptable value (the closest edge of the small cube). See Figure 24-49.

Figure 24-49 How minimum and maximum color limits affect drawing



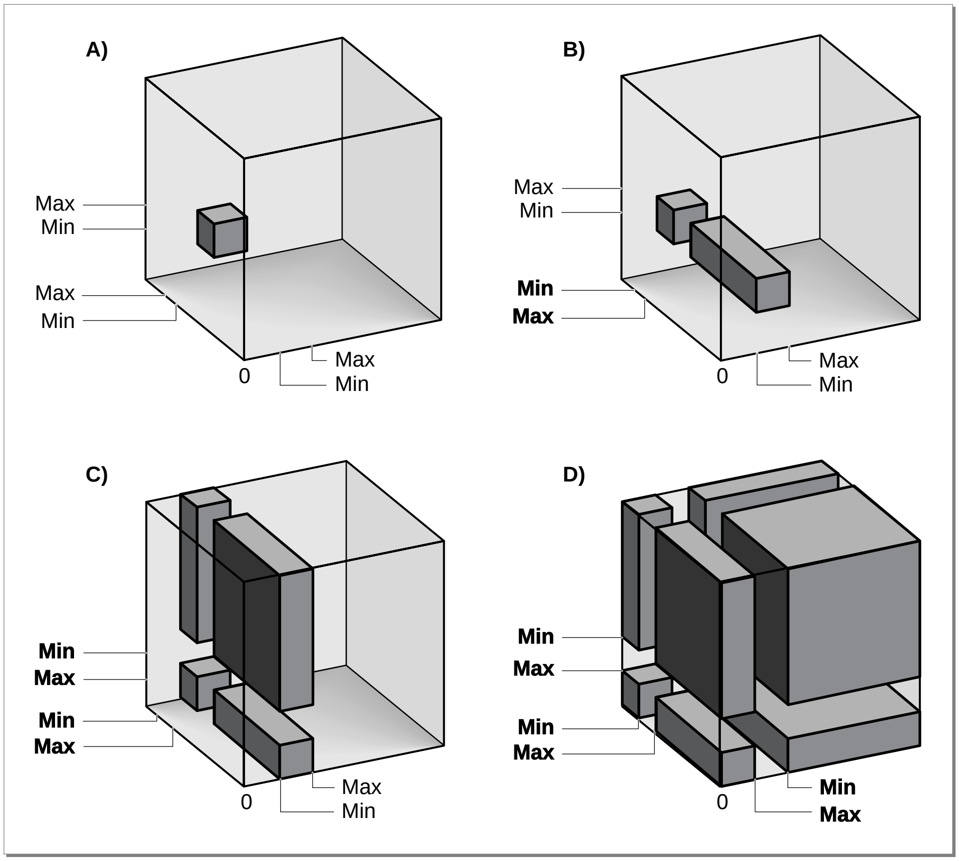
For a given component, the maximum value for a color limit can be either greater or smaller than the minimum. If the maximum is less than the minimum, only the extreme color values (that is, values *outside* of the small cube area in Figure 24-48) are allowed. See Figure 24-50.

Figure 24-50 How reversed minimum and maximum color limits affect drawing



Each of the components in a color space can have its limits set entirely independently of the others. Figure 24-51 shows the effects of reversing, in turn, the maximum and minimum values for each of the three axes in RGB space.

Figure 24-51 The effects of reversing maximum and minimum in a color space



Where the words *Min* and *Max* are bold in Figure 24-51, the minimum is greater than the maximum. Refer to Figure 24-48 (page 835) for the positions of the color axes on the RGB cube in this figure:

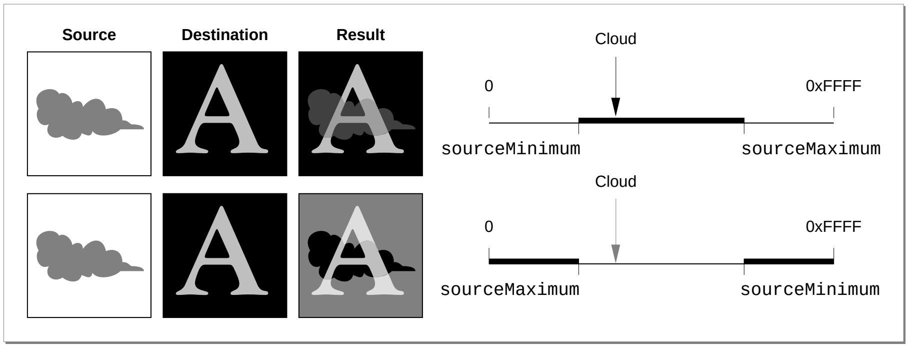
- In drawing (A), all minimum limits are less than their respective maximums; the allowable color ranges form a small cube, just as in Figure 24-48.
- In drawing (B), the maximum on the red axis is less than the minimum; only red color values outside of the range of the small cube are permitted, whereas blue and green must still be within the limits of the small cube. The acceptable color values form two rectangular solids within RGB space.

- In drawing (C), the maximum and minimum on the green axis are also reversed; the acceptable color values form a more complicated set of solids.
- In drawing (D), all maximum and minimum color limits are reversed. In that case, only color values at the outer corners of the color space (all components outside of the range of the small cube) are acceptable.

Source Color Limits

The `sourceMinimum` and `sourceMaximum` fields in a color component's `gxTransferComponent` structure define the allowable range of values for source color in that component. Color values outside of the range cause no drawing to occur. If `sourceMaximum` is less than `sourceMinimum`, the range allowed consists of values less than `sourceMaximum` or greater than `sourceMinimum`. Figure 24-52 shows the effect of `sourceMinimum` and `sourceMaximum` on drawing using blend mode.

Figure 24-52 The effect of source color limits on drawing

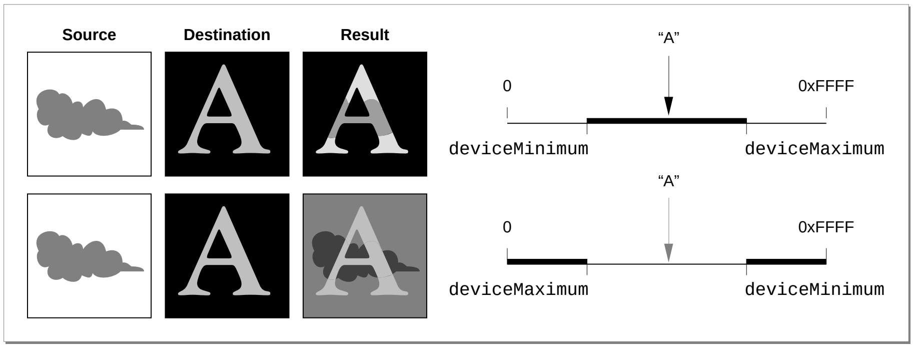


Note in Figure 24-52 that, when `sourceMinimum` is less than `sourceMaximum`, only the cloud in the source image is within the source limits, so only the cloud is blended with the destination image to create the result. Conversely, when `sourceMaximum` is less than `sourceMinimum`, the cloud in the source image is outside the source limits, so it is the only part of the source that is *not* blended with the destination image when creating the result.

Destination Color Limits

The `deviceMinimum` and `deviceMaximum` fields in a color component's `gxTransferComponent` structure define the allowable range of values for destination color in that component. Destination color values outside of the range cause no drawing to occur for that color. If `deviceMaximum` is less than `deviceMinimum`, the range allowed consists of values less than `deviceMaximum` or greater than `deviceMinimum`. Figure 24-53 shows the effect of `deviceMinimum` and `deviceMaximum` on drawing using blend mode.

Figure 24-53 The effect of destination color limits on drawing



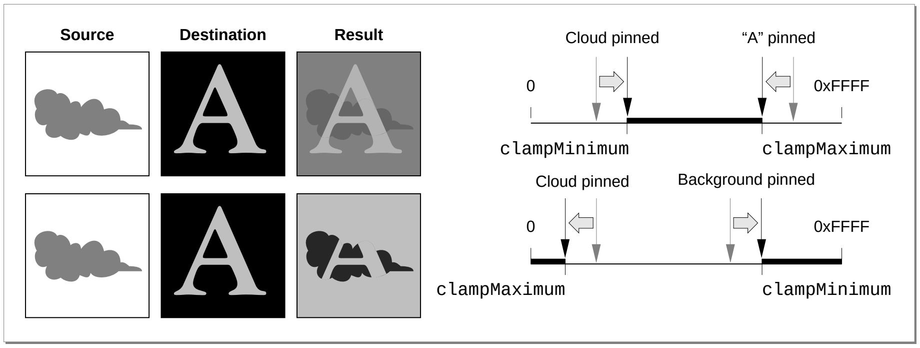
Note in Figure 24-53 that, when `deviceMinimum` is less than `deviceMaximum`, only the letter "A" in the destination image is within the destination limits, so the source is blended with the destination image only within the limits of the "A" to create the result. Conversely, when `deviceMaximum` is less than `deviceMinimum`, the "A" is outside the destination limits, so it is the only part of the destination *not* blended with the source to create the result.

Result Color Limits

The `clampMinimum` and `clampMaximum` fields in a color component's `gxTransferComponent` structure define the allowable range of values for the result color in that component. Color values outside of the range are pinned to the nearest clamp limit. If `clampMaximum` is less than `clampMinimum`, the range allowed consists of values less than `clampMaximum` or greater than `clampMinimum`.

Figure 24-54 shows the effect of `clampMinimum` and `clampMaximum` on drawing using blend mode.

Figure 24-54 The effect of result color limits on drawing



Note in Figure 24-54 that, when `clampMinimum` is less than `clampMaximum`, extreme color values cannot occur in the result. The portions of the "A" outside of the cloud are darker than they would normally be with blend mode, and the portions of the cloud outside of the letter are lighter than they would normally be. Conversely, when `clampMaximum` is less than `clampMinimum`, midrange values are not possible in the result. The background in the result is lighter than it would normally be with blend mode, and the portions of the cloud outside of the "A" are darker than they would normally be.

Note
Pinning restricts the value of the computation, not necessarily the value allowed for the actual pixel. The pixel value is the closest found to the computation, which may be outside of the range of `clampMinimum` and `clampMaximum`. ◆

Transfer Mode Matrices

QuickDraw GX provides three matrices in the transfer mode structure to give you great freedom in controlling, modifying, and combining source, destination, and result color components when performing a transfer mode operation.

The **source matrix, device matrix,** and **result matrix** provide a way of scaling, weighting, swapping, and averaging the components of a color space before or after the transfer mode operation. Each matrix is a 5×4 array that specifies the mixture of each of the (up to 4) components, plus an offset.

An **identity matrix,** one that has values of 1.0 along the diagonal and zero values elsewhere, has no effect. Here it is applied to a color in CMYK space:

$$\begin{bmatrix} c & m & y & k & 1 \end{bmatrix} \begin{bmatrix} 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \\ 0.0 & 0.0 & 0.0 & 0.0 \end{bmatrix} = \begin{bmatrix} c & y & m & k \end{bmatrix}$$

The values for all color components after the matrix multiplication are the same as before. All transfer mode matrices in the default ink object are identity matrices.

The bottom row of the matrix specifies an offset value. The following matrix replaces c with 1/2 c + 1/2 m; it also scales k by 0.8 and adds 0.2 to it:

$$\begin{bmatrix} c & m & y & k & 1 \end{bmatrix} \begin{bmatrix} 0.5 & 0.0 & 0.0 & 0.0 \\ 0.5 & 1.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.8 \\ 0.0 & 0.0 & 0.0 & 0.2 \end{bmatrix} = \begin{bmatrix} (0.5c + 0.5m) & m & y & (0.8k + 0.2) \end{bmatrix}$$

The source and device matrix are applied before the transfer mode calculation and after applying source minimum and source maximum. The result of the transfer mode calculation is run through the result matrix. The use of matrices allows you to apply sophisticated mapping operations—analogous to the scaling, rotation, translation, and distortion of shapes—to the colors involved in a transfer mode operation. Matrices are also used to create color separations, and to map source color ranges to spot colors.

Note
Although color components are described by unsigned short integers (16-bit positive numbers), the math internal to transfer modes is performed with long integers (32-bit signed numbers) to minimize overflow or roundoff error. As an example, elements in the source matrix could multiply by a large number, and elements in the result matrix could divide by a large number, without creating an overflow condition. ◆

Flags for Transfer Modes

QuickDraw GX provides two sets of flags in the transfer mode structure that control certain aspects of the transfer mode operation. One set operates on individual color components; the other set operates on the source or destination color as a whole, taking into account all of the components.

Transfer Component Flags

The transfer component flags are a set of flags in the `gxTransferComponent` structure (in the `component` field of the transfer mode structure) that alter the source, destination, or result value for an individual color component. There are two constants for these flags, defined in the `gxComponentFlags` enumeration:

Constant	Value	Explanation
`gxOverResultComponent`	0x01	QuickDraw GX performs an AND operation between the result color and 0xFFFF before clamping.
`gxReverseComponent`	0x02	QuickDraw GX reverses the source and destination values before performing the transfer mode operation.

Specifying `gxOverResultComponent` allows the result of transfers using `gxAddMode` to wrap around (from `0xFFFF` to `0x0000`) instead of remaining clamped at `0xFFFF`.

Specifying `gxReverseComponent` allows you to apply a transfer mode backwards—from the destination to the source—for a particular component. It is most useful for component modes that depend on order, like `gxMigrateMode`, or `gxAddMode` when used for subtraction.

Transfer Mode Flags

The transfer mode flags are a set of flags in the `flags` field of the transfer mode structure. They affect how color limits are used and whether a single component mode is to be used for all color components. There are three values for the flags, defined in the `gxTransferFlags` enumeration:

Constant	Value	Explanation
`gxRejectSourceTransfer`	0x0001	Negate the results of `sourceMinimum` and `sourceMaximum` for all components. Accept only values *outside* of the specified ranges.
`gxRejectDeviceTransfer`	0x0002	Negate the results of `deviceMinimum` and `deviceMaximum` for all components. Accept only values *outside* of the specified ranges.
`gxSingleComponentTransfer`	0x0004	Use a single transfer component for all color components. Duplicate `component[0]` in the transfer mode structure for all components in the transfer mode's color space.

Setting the `gxRejectSourceTransfer` or `gxRejectDeviceTransfer` flag causes an inversion of the acceptable color ranges for source or destination color, respectively. For example, in Figure 24-51 (page 837), setting the `gxRejectSourceTransfer` or `gxRejectDeviceTransfer` flag would cause the white (empty) portions of the large cubes that represent RGB space to be within range, instead of the gray (filled) portions.

The effect is similar to, although not exactly the same as, individually reversing the minimum and maximum values for the color components. If the transfer mode flag is cleared, drawing occurs only when *all* components are inside the allowed ranges—that is, inside the darker gray portions of the large cubes in Figure 24-51. With the flag set, drawing occurs any time *at least one* component is outside of its allowed range—that is, with values anywhere outside of the dark gray areas in Figure 24-51.

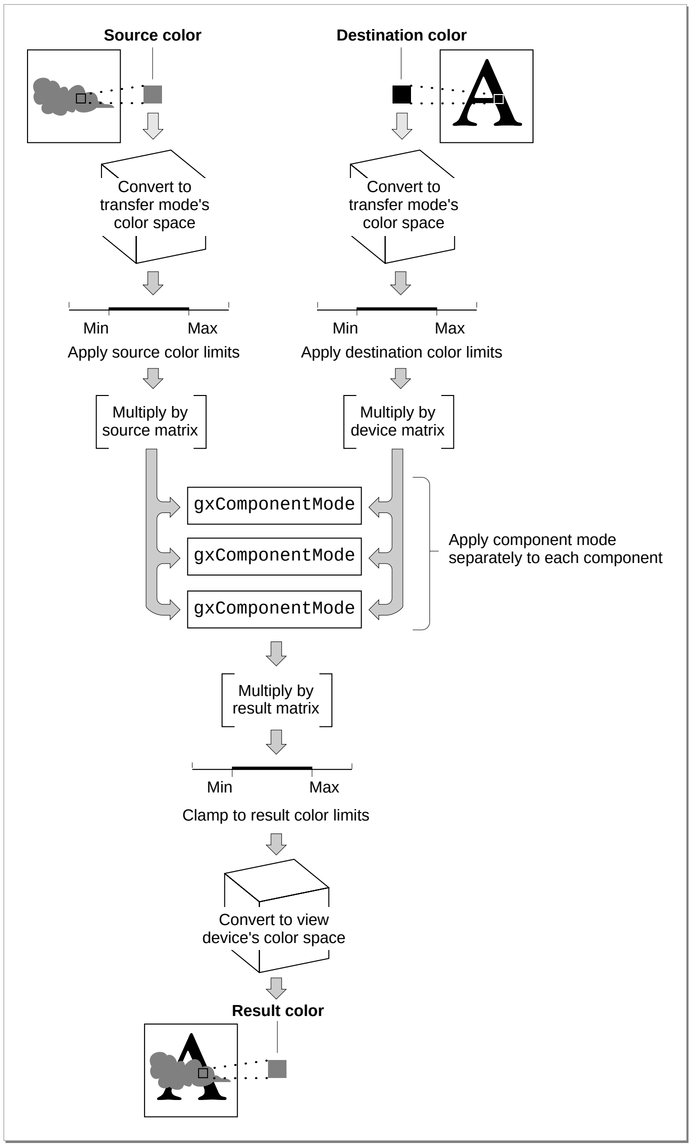
The `gxSingleComponentTransfer` flag is provided as a convenience. You can set the flag when you don't need the flexibility (and extra effort) of specifying different transfer modes for different color components. In this case you need set up only one `gxTransferComponent` structure, instead of one for each component in the transfer mode's color space.

Summary of Transfer Mode Operation

Figure 24-55 shows how all the parts of the transfer mode structure work together when a color is drawn.

1. The source color is converted to the transfer mode's color space, if they are not already the same. Each component of the source color is then compared to the acceptable range of source colors, modified if appropriate by the values of the transfer mode flags. The resulting source components are then multiplied by the source matrix to yield the corrected source components for the transfer mode operation.
2. The destination color is converted to the transfer mode's color space, if necessary. Each component of the destination color is then compared to the acceptable range of destination colors, modified if appropriate by the values of the transfer mode flags. The resulting destination components are then multiplied by the device matrix to yield the corrected destination components for the transfer mode operation.
3. The pairs of source and destination components are each combined, according to the selected component mode, and the value of the operand if the component mode takes one. (Source and destination values for a component are swapped before combining if the `gxReverseComponent` component flag is set.)
4. The components that result from the transfer mode operation are each multiplied by the results matrix to yield the corrected result components (and then ANDed with `gxColorValue1` (`0xFFFF`) if the `gxOverResultComponent` flag is set for that component.) Each component is then pinned, if necessary, to the acceptable range of result colors. Finally, the result color is converted to the color space of the view device, and the color is drawn.

Figure 24-55 Summary of transfer mode operation


Source color
Destination color
Convert to transfer mode's color space
Convert to transfer mode's color space
Min
Max
Apply source color limits
Min
Max
Apply destination color limits
Multiply by source matrix
Multiply by device matrix
gxComponentMode
gxComponentMode
gxComponentMode
Apply component mode separately to each component
Multiply by result matrix
Min
Max
Clamp to result color limits
Convert to view device's color space
Result color

QuickTime Vector Data Streams

In QuickTime, paths and their characteristics are represented by a series of atoms in a QT atom container. This ordered series of atoms is called a **vector data stream.** For more information about QT atom containers, see “QuickTime Atoms” (page 47).

A vector data stream contains atoms for paths, atoms that specify attributes of paths, and a final atom that marks the end of the data stream. To draw paths, QuickTime traverses the atoms in a vector data stream in order. When it finds an atom for a path, it draws the path using the current attribute values. When it finds an atom for an attribute, the new attribute value replaces the previous value of the attribute. If QuickTime has not found an atom for a particular attribute before drawing a path, it uses the default value for the attribute.

Within a vector data stream, all values must be aligned on long-word boundaries and stored in big-endian format. Values that you add using vector component functions automatically have the proper alignment and format. For descriptions of these functions, see “Vector Codec Component Functions” (page 895).

Contents of a Vector Data Stream

The types of atoms that can be included in a vector data stream are summarized in the following sections. For complete descriptions of these atoms, see “Vector Atom Types” (page 871).

Required Atoms

A QuickTime vector data stream must contain at least two types of atoms:

- One or more `kCurvePathAtom` atoms, each of which specifies a path to draw.

  Each path can contain up to 32767 contours, each of which may contain up to 32767 points. Each point may be located on or off the curve, with the curve describing a quadratic Bezier. For an overview of paths, see “Path Shapes” (page 771).
- A `kCurveEndAtom` atom that marks the end of the vector data stream.

There can be only one `kCurveEndAtom` atom in a vector data stream, and it must be the last atom in the stream.

Atoms that Specify Path Attributes

Atoms that specify path attributes are optional. If the QuickTime vector codec has not found an atom in the data stream for a particular path attribute, it uses the default value for that attribute when drawing paths. If it does find an atom for a particular attribute, it uses its value when drawing subsequent paths in the data stream until the value is replaced by another atom of the same type or the end of the data stream is reached.

With one exception, any atom that specifies a path attribute can appear anywhere in a vector data stream, and there can be any number of atoms of the same type. The exception is the `kCurveMinimumDepthAtom` atom that specifies the minimum bit depth used to draw paths: there can be at most one `kCurveMinimumDepthAtom` atom in a vector data stream, and if included it must be the first atom in the stream.

These are the atoms that specify path attributes:

- A `kCurveAntialiasControlAtom` atom enables or disables anti-aliasing for subsequent paths in the data stream. Anti-aliasing is a technique in which edges of the path that do not fall on pixel boundaries are be drawn by alpha blending the color of the path with the color of the background. The alpha values take into account errors introduced by the pixelization of the shape and give edges a smoother appearance.
- A `kCurveARGBColorAtom` atom specifies the color of subsequent paths in the data stream.
- A `kCurveFillTypeAtom` atom specifies the fill type for subsequent paths in the data stream. For more information about fill types, see "Shape Fill" (page 764).
- `kCurveGradientAngleAtom`, `kCurveGradientOffsetAtom`, `kCurveGradientRadiusAtom`, `kCurveGradientRecordAtom`, and `kCurveGradientTypeAtom` atoms specify attributes of gradient fills for subsequent paths in the data stream. Gradient fills are described in "Gradients for Path Fills" (page 848).
- A `kCurveJoinAttributesAtom` atom specifies the type of join for subsequent paths in the vector data stream with sharp angles, and a

`kCurveMiterLimitAtom` atom specifies the miter limit of joins, as described in "Joins" (page 786).

- A `kCurveMinimumDepthAtom` atom specifies the minimum bit depth for drawing subsequent paths in the vector data stream. There can be at most one `kCurveMinimumDepthAtom` atom in a vector data stream, and if included it must be the first atom in the stream.
- A `kCurvePenThicknessAtom` atom specifies the thickness of the stroke for subsequent framed paths in the vector data stream, as described in "Pen Placement" (page 784).
- A `kCurveTransferModeAtom` specifies the transfer mode for drawing paths. For information about transfer modes, see "Transfer Modes" (page 814).

QuickTime Vector Features Not Included in QuickDraw GX

QuickTime includes two features that are not included in QuickDraw GX: gradient fills for paths and the ability to specify a minimum bit depth for paths. These features are described in the following sections.

Gradients for Path Fills

A **gradient** is a fill that contains continuous fades from one color to another. Unlike QuickDraw GX, which supports gradient fills only for bitmaps, QuickTime supports gradient fills for the paths used for QuickTime vectors.

There are two types of gradients: linear, in which the fade begins at one point on a line and ends at another, and circular, in which the fade begins at the center of a circle and ends at the circumference. Figure 24-56 illustrates a white-to-black, left-to-right linear gradient, and Figure 24-57 illustrates a white-to-black circular gradient.

Figure 24-56 A linear gradient

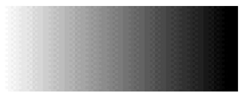
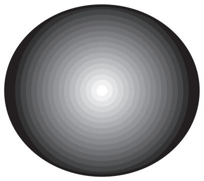


Figure 24-57 A circular gradient



To enable or disable gradient fills and to specify the attributes of gradients, you add atoms to the vector data stream, as described in “Using QuickTime Vectors” (page 850):

- You specify the type of gradient (either linear or circular) with a `kCurveGradientTypeAtom` atom.
- For linear gradients, you can specify the direction in which the gradient is drawn with a `kCurveGradientAngleAtom` atom. You specify this direction with an angle: for example, 0.0 specifies a top-to-bottom gradient, 90.0 specifies a right-to-left gradient, 180.0 specifies a bottom-to-top gradient, and 270.0 specifies a left-to-right gradient.
- For circular gradients, you can specify the radius of the gradient with a `kCurveGradientRadiusAtom` atom.
- You specify the colors for gradients with a `kCurveGradientRecordAtom` atom. A gradient must have at least two colors, for the beginning and end, but it can also any number of intermediate colors. Each color is specified by a `GradientColorRecord` structure. You can also add a `kCurveGradientRecordAtom` atom without data to disable gradient fills.
- You can specify an offset for gradients with a `kCurveGradientOffsetAtom` atom. For circular gradients, this specifies the top left offset from the path being drawn. For linear gradients, it is the offset from the center of the rotated gradient line.

For complete descriptions of these atoms and their default values, see “Vector QT Atom Container” (page 874).

Specifying the Bit Depth for Paths

QuickTime allows you specify the minimum bit depth for drawing all paths in a vector data stream. If the bit depth of the display device is less than the bit depth specified by the `kCurveMinimumDepthAtom` atom, the path is rendered into an offscreen buffer with the minimum bit depth and then transferred to the screen. This value must be at least 16 for most transfer modes. If the quality of a dithered image is unsatisfactory, increasing the `kCurveMinimumDepthAtom` value to 32 may improve the result.

A `kCurveMinimumDepthAtom` atom is optional. By default, the bit depth of the output is not changed. There can be at most one `kCurveMinimumDepthAtom` atom in a vector data stream, and if included it must be the first atom in the stream.

You may also specify `kCurveDepthAlwaysOffscreenMask` to force rendering offscreen of complex curves.

Using QuickTime Vectors

This section explains how to create and manipulate QuickTime vectors. The examples use utility functions provided by the vector codec, described in "Vector Codec Component Functions" (page 895), that eliminate the need to work directly with the QuickTime atoms in a vector data stream. If your software edits or parses a great deal of vector data, it may be more efficient to work directly with the atoms. For descriptions of the atoms in a vector data stream, see "Vector QT Atom Container" (page 874). For information about working directly with QuickTime atoms, see "QuickTime Atoms" (page 47).

There are several issues concerning the creation of curve movies that developers should be aware of. First, the most common mistake that developers may make when authoring a curve movie is to fail to place an "eraser shape" (usually white) of some kind behind the main shape. If an eraser shape is not placed behind the main shape, odd behaviors may occur:

- Resizing / updating the window may result in the old size not being erased or "overlapped"
- Curves that feature transfer modes blend and re-blend across themselves.

The second issue involves placing samples on top of another track (such as a movie). This works properly in only two cases:

- If the user selects “composite mode” for the graphics mode of the curve track
- If users place a white eraser rectangle and then use transparent mode.

Proper use of curves can produce QuickTime movies with spectacular effects, such as a red- blended curve with a video running underneath it. However, failure to use the proper mode may often result in confusion for the user.

Note that many of these problems do *not* occur when the user uses curves in a sprite track rather than directly in a video track.

Opening the Vector Codec Component

The utility functions you use to create and manipulate vector data streams are component functions of the vector codec (also known as the Apple curve codec). This codec, which processes vector data streams and draws the paths they specify, is a standard component that is managed by the Component Manager. Before calling these functions, you must first open the codec as shown in Listing 24-1.

Listing 24-1 Opening the vector codec

```
ComponentInstance vectorCodec;
OpenADefaultComponent (decompressorComponentType,
                                kVectorCodecType, &vectorCodec);
```

For more information about the Component Manager and using components, see the chapter “Component Manager” in *Mac OS For QuickTime Programmers*.

Creating a Vector Data Stream

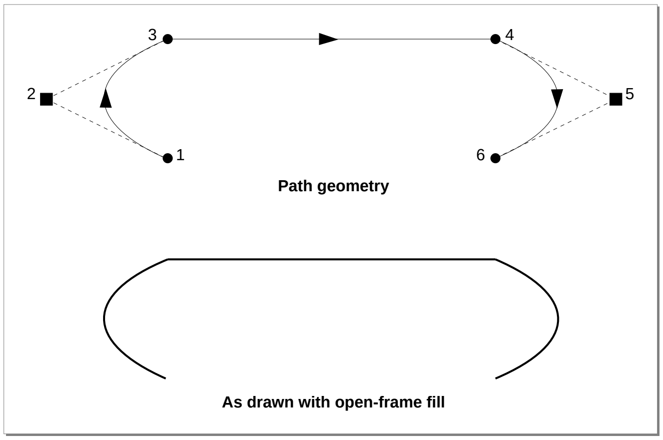
Listing 24-2 shows how to create a vector data stream for a single path. The path has four on-curve points and two off-curve points. These points specify two curves and one straight line, as shown in Figure 24-58. In addition to an atom for the path, the data stream includes an atom that specifies the fill type of subsequent paths in stream.

Listing 24-2 vector data stream for a single path

```
ComponentInstance vectorCodec;
Handle streamH;
Handle pathH;
gxPoint points[6];
Boolean isOnCurve[6];
int i;
long value;

/* open the vector codec component */
OpenADefaultComponent (decompressorComponentType,
                        kVectorCodecType, &vectorCodec);

/* create a new vector data stream */
CurveCreateVectorStream (vectorCodec, streamH);
value=gxOpenFrameFill;

/* add an atom to the vector data stream that specifies that
   subsequent paths are to be drawn with open frame fill */
CurveAddAtomToVectorStream (ci, kCurveFillTypeAtom, sizeof(long),
                            &value, streamH);

/* specify the points for the path and whether each one is
   on the path */
points[0].x = ff(50);
points[0].y = ff(100);
isOnCurve[0] = TRUE;
points[1].x = ff(0);
points[1].y = ff(75);
isOnCurve[1] = FALSE;
points[2].x = ff(50);
points[2].y = ff(50);
isOnCurve[2] = TRUE;
points[3].x = ff(150);
points[3].y = ff(50);
isOnCurve[3] = TRUE;
points[4].x = ff(200);
points[4].y = ff(75);
isOnCurve[4] = FALSE;
points[5].x = ff(150);
```

```
points[5].y = ff(100);
isOnCurve[5] = TRUE;

/* create the path and add the points to it */
CurveNewPath (vectorCodec, &pathH);
for (i = 0; i <= 5; i++)
    CurveInsertPointIntoPath (vectorCodec, &points[i], pathH,
                             1, i, isOnCurve[i]);

/* add the path to the vector data stream */
CurveAddPathAtomToVectorStream (vectorCodec, pathH, streamH);

/* mark the end of the vector data stream by adding a
   kCurveEndAtom atom to the stream */
CurveAddZeroAtomToVectorStream (vectorCodec, streamH);

/* use the vector codec here to decompress and display the vector data */

/* dispose of stream and path handles when done */
DisposeHandle (streamH);
DisposeHandle (pathH);
```

Figure 24-58 A path with six points



Creating a Path Using Only Off-Curve Points

Listing 24-3 shows how you can create a path using only off-curve control points. The path defined in this example contains four control points.

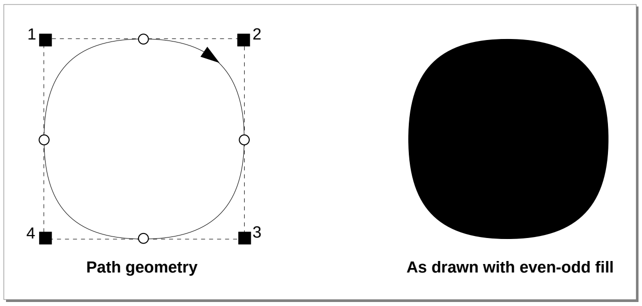
Listing 24-3 Creating a path using only off-curve control points

```
ComponentInstance vectorCodec;
Handle streamH;
Handle pathH;
gxPoint points[4];
Boolean isOnCurve[4];
int i;

/* open the vector codec component */
OpenADefaultComponent (decompressorComponentType,
                        kVectorCodecType, &vectorCodec);

/* create a new vector data stream */
CurveCreateVectorStream (vectorCodec, streamH);

/* specify the points for the path and whether each one is on it */
points[0].x = ff(50);
points[0].y = ff(50);
isOnCurve[0] = FALSE;
points[1].x = ff(150);
points[1].y = ff(50);
isOnCurve[1] = FALSE;
points[2].x = ff(150);
points[2].y = ff(150);
isOnCurve[2] = FALSE;
points[3].x = ff(50);
points[3].y = ff(150);
isOnCurve[3] = FALSE;

/* create the path and add the points to it */
CurveNewPath (vectorCodec, &pathH);
for (i = 0; i <= 3; i++)
    CurveInsertPointIntoPath (vectorCodec, &points[i], pathH,
                              1, i, isOnCurve[i]);
```

```
/* add the path to the vector data stream */
CurveAddPathAtomToVectorStream (vectorCodec, pathH, streamH);

/* mark the end of the vector data stream by adding a
   kCurveEndAtom atom to the stream */
CurveAddZeroAtomToVectorStream (vectorCodec, streamH);

/* use the vector codec here to decompress and display the vector data */

/* dispose of stream and path handles when done */
DisposeHandle (streamH);
DisposeHandle (pathH);
```

The four off-curve control points in this example form a square; the path that they define is a rounded square, as shown in Figure 24-59.

Figure 24-59 A rounded path shape



Notice that the path is filled with the even-odd shape fill, which is the default for path shapes. You could, however, specify any shape fill for this path except the open-frame shape fill. The open-frame shape fill requires that the first and last points of the contour be on-curve points, and this path has no on-curve points.

Creating Paths With Multiple Contours and Fills

Listing 24-4 shows how a single path shape can contain more than one path contour. The path shape defined in this example includes the round path from the previous example as well as a second round path, entirely contained within the first.

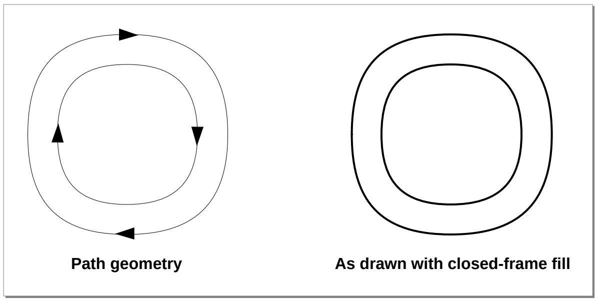
Listing 24-4 Creating a path with concentric contours

```
ComponentInstance vectorCodec;
Handle streamH;
Handle pathH;
gxPoint points[4];
Boolean isOnCurve[4];
int i;
long value;

/* open the vector codec component */
OpenADefaultComponent (decompressorComponentType,
                        kVectorCodecType, &vectorCodec);

/* create a new vector data stream */
CurveCreateVectorStream (vectorCodec, streamH);
value=gxClosedFrameFill

/* add an atom to the vector data stream that specifies that
   subsequent paths are to be drawn with closed frame fill */
CurveAddAtomToVectorStream (ci, kCurveFillTypeAtom,
                            sizeof(long), &value, streamH);

/* create the path */
CurveNewPath (vectorCodec, &pathH);

/* specify the points for the first contour and whether each one is
   on the path */
points[0].x = ff(50);
points[0].y = ff(50);
isOnCurve[0] = FALSE;
points[1].x = ff(150);
points[1].y = ff(50);
```

```
isOnCurve[1] = FALSE;
points[2].x = ff(150);
points[2].y = ff(150);
isOnCurve[2] = FALSE;
points[3].x = ff(50);
points[3].y = ff(150);
isOnCurve[3] = FALSE;

/* add the points for the first contour to the path */
for (i = 0; i <= 3; i++)
    CurveInsertPointIntoPath (vectorCodec, &points[i], pathH,
                              1, i, isOnCurve[i]);

/* specify the points for the second contour and whether each one is
   on the path */
points[0].x = ff(65);
points[0].y = ff(65);
isOnCurve[0] = FALSE;
points[1].x = ff(135);
points[1].y = ff(65);
isOnCurve[1] = FALSE;
points[2].x = ff(135);
points[2].y = ff(135);
isOnCurve[2] = FALSE;
points[3].x = ff(65);
points[3].y = ff(135);
isOnCurve[3] = FALSE;

/* add the points for the second contour to the path */
for (i = 0; i <= 3; i++)
    CurveInsertPointIntoPath (vectorCodec, &points[i], pathH,
                              2, i, isOnCurve[i]);

/* add the path to the vector data stream */
CurveAddPathAtomToVectorStream (vectorCodec, pathH, streamH);

/* mark the end of the vector data stream by adding a
   kCurveEndAtom atom to the stream */
CurveAddZeroAtomToVectorStream (vectorCodec, streamH);

/* use the vector codec here to decompress and display the vector data */
/* dispose of stream and path handles when done */
DisposeHandle (streamH);
DisposeHandle (pathH);
```

The result is shown in Figure 24-60.

Figure 24-60 A path with two concentric clockwise contours and closed-frame fill

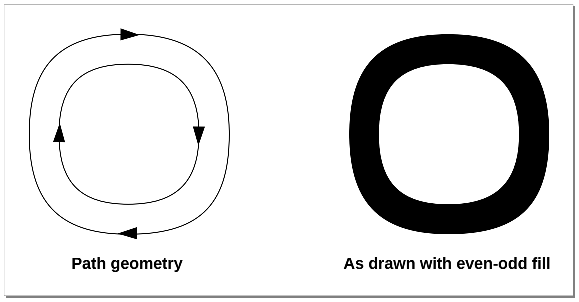


You can change the shape fill for this path by removing these lines from Listing 24-4:

```
/* add an atom to the vector data stream that specifies that
   subsequent paths are to be drawn with closed frame fill */
CurveAddAtomToVectorStream (ci, kCurveFillTypeAtom, sizeof(long),
                            &value, streamH);
```

If you don’t specify a fill type, you get the default fill type, which is the even-odd fill. The path shape resulting from an even-odd fill is shown in Figure 24-61.

Figure 24-61 A path with two contours and even-odd shape fill

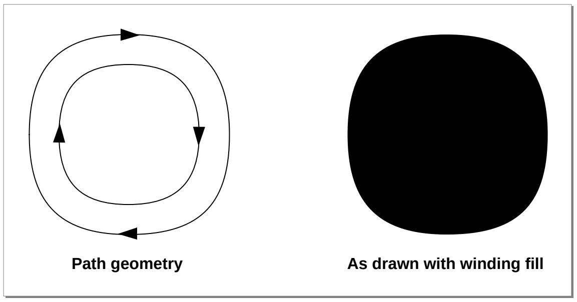


Notice that the even-odd shape fill causes the vector codec to fill in the outer contour, but not the inner contour. You can specify a winding fill by adding the following code before the code to draw a path:

```
/* add an atom to the vector data stream that specifies that
   subsequent paths are to be drawn with winding fill */
CurveAddAtomToVectorStream (ci, kCurveFillTypeAtom, sizeof(long),
                            &value, streamH);
```

The path drawn with a winding fill is shown in Figure 24-62.

Figure 24-62 A path with two contours and winding fill



Unlike the even-odd shape fill, the winding shape fill causes the vector codec to fill inner contours—*as long as the inner contour has the same contour direction as the outer contour.* If the inner contour and the outer contour have opposite contour directions, neither the even-odd shape fill nor the winding shape fill will fill the inner contour.

For example, if you change the direction of the inner contour from the previous example by reversing the order of the second contour's geometric points, as in the following code

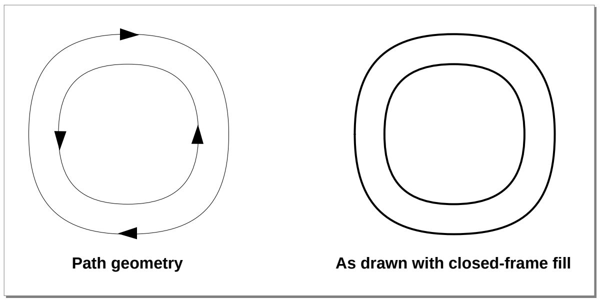
```
/* specify the points for the second contour and whether each one is
   on the path */
points[0].x = ff(65);
points[0].y = ff(135);
isOnCurve[0] = FALSE;
points[1].x = ff(135);
points[1].y = ff(135);
isOnCurve[1] = FALSE;
points[2].x = ff(135);
points[2].y = ff(65);
isOnCurve[2] = FALSE;
points[3].x = ff(65);
points[3].y = ff(65);
isOnCurve[3] = FALSE;

/* add the points for the second contour to the path */
for (i = 0; i <= 3; i++)
    CurveInsertPointIntoPath (vectorCodec, &points[i], pathH,
                              2, i, isOnCurve[i]);
```

and you specify a closed-frame fill by adding the following code before the code to draw a path:

```
/* add an atom to the vector data stream that specifies that
   subsequent paths are to be drawn with winding fill */
CurveAddAtomToVectorStream (ci, kCurveFillTypeAtom, sizeof(long),
                            &value, streamH);
```

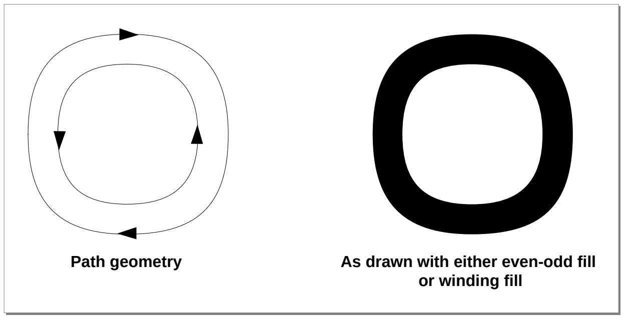
the resulting path has contours with opposite contour directions, as depicted in Figure 24-63.

Figure 24-63 Path with internal counterclockwise contour and closed-frame fill



Since the outer contour and the inner contour have opposite contour directions, neither the even-odd shape fill nor the winding shape fill cause the vector codec to fill the inner contour, as shown in Figure 24-64.

Figure 24-64 A path with even-odd or winding shape fill



Specifying Joins

You specify attributes of joins by including `kCurveJoinAttributesAtom` and `kCurveMiterLimitAtom` atoms in a vector data stream. The `kCurveJoinAttributesAtom` atom specifies the type of join, and the `kCurveMiterLimitAtom` specifies the miter limit for joins. For more information about join types and miter limits, see “Joins” (page 786).

As with other atoms that specify path attributes, an attribute value specified by a `kCurveJoinAttributesAtom` or `kCurveMiterLimitAtom` atom is used for subsequent paths in the vector data stream until another atom of the same type appears in the stream or the end of the stream is reached.

Listing 24-5 shows how to specify a sharp join for an angle.

Listing 24-5 Adding a sharp join to an angle

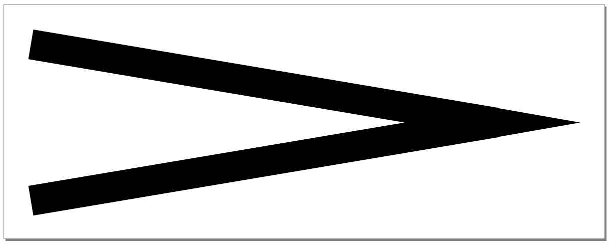
```
ComponentInstance vectorCodec;
Handle streamH;
Handle pathH;
gxPoint points[3];
Boolean isOnCurve[3];
int i;
long value;

/* open the vector codec component */
OpenADefaultComponent (decompressorComponentType,
                        kVectorCodecType, &vectorCodec);

/* create a new vector data stream */
CurveCreateVectorStream (vectorCodec, streamH);
value=gxOpenFrameFill;
value=ff(15);
value=gxSharpJoin;
value=gxPositiveInfinity;

/* add an atom to the vector data stream that specifies that
   subsequent paths are to be drawn with open frame fill */
CurveAddAtomToVectorStream (ci, kCurveFillTypeAtom, sizeof(long),
                            &value, streamH);

/* add an atom to the vector data stream that specifies that
   subsequent paths are to be drawn with a pen width of 15 */
CurveAddAtomToVectorStream (ci, kCurvePenThicknessAtom, sizeof(Fixed),
                            &value, streamH);

/* add an atom to the vector data stream that specifies that
   subsequent paths are to be drawn with sharp angle joins*/
```

```
CurveAddAtomToVectorStream (ci, kCurveJoinAttributesAtom,
                            sizeof(gxJoinAttribute),
                            &value, streamH);

/* add an atom to the vector data stream that specifies that
   subsequent paths are to have no miter limit for joins */
CurveAddAtomToVectorStream (ci, kCurveMiterLimitAtom, sizeof(Fixed),
                            &value, streamH);

/* specify the points for the path and whether each one is
   on the path */
points[0].x = ff(20);
points[0].y = ff(20);
isOnCurve[0] = TRUE;
points[1].x = ff(250);
points[1].y = ff(60);
isOnCurve[1] = FALSE;
points[2].x = ff(20);
points[2].y = ff(100);
isOnCurve[2] = TRUE;

/* create the path and add the points to it */
CurveNewPath (vectorCodec, &pathH);
for (i = 0; i <= 2; i++)
    CurveInsertPointIntoPath (vectorCodec, &points[i], pathH,
                              1, i, isOnCurve[i]);

/* add the path to the vector data stream */
CurveAddPathAtomToVectorStream (vectorCodec, pathH, streamH);

/* mark the end of the vector data stream by adding a
   kCurveEndAtom atom to the stream */
CurveAddZeroAtomToVectorStream (vectorCodec, streamH);

/* use the vector codec here to decompress and display the vector data */

/* dispose of stream and path handles when done */
DisposeHandle (streamH);
DisposeHandle (pathH);
```

Notice that miter limit is set to the constant value `gxPositiveInfinity`, which indicates the join should be as sharp as necessary.

Figure 24-65 shows what is drawn for this vector data stream.

Figure 24-65 An angle with a sharp join

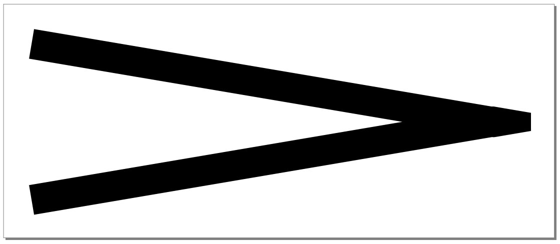


If you limit the miter of the sharp join, for example, with the code

```
/* add an atom to the vector data stream that specifies that
   subsequent paths are to have no miter limit for joins */
CurveAddAtomToVectorStream (ci, kCurveMiterLimitAtom, sizeof(Fixed),
                            &value, streamH);
```

the vector codec limits the distance between the actual corner of the contour as specified in the shape's geometry and the tip of the corner as actually drawn. Since the miter is scaled by the pen width, and the pen width in this example is 15, the vector codec truncates the sharp join 15 points away from the actual corner of the geometry, as shown in Figure 24-66.

Figure 24-66 An angle with a truncated sharp join



Adding Gradients

You specify gradient fills and their attributes by including these atoms in a vector data stream:

- You specify the type of gradient (either linear or circular) with a `kCurveGradientTypeAtom` atom.
- For linear gradients, you can specify the direction in which the gradient is drawn with a `kCurveGradientAngleAtom` atom. You specify this direction with an angle: for example, 0.0 specifies a top-to-bottom gradient, 90.0 specifies a right-to-left gradient, 180.0 specifies a bottom-to-top gradient, and 270.0 specifies a left-to-right gradient.
- For circular gradients, you can specify the radius of the gradient with a `kCurveGradientRadiusAtom` atom.
- You specify the colors for gradients with a `kCurveGradientRecordAtom` atom. A gradient must have at least two colors, for the beginning and end, but it can also any number of intermediate colors. Each color is specified by a `GradientColorRecord` structure. You can also add a `kCurveGradientRecordAtom` atom without data to disable gradient fills.
- You can specify an offset for gradients with a `kCurveGradientOffsetAtom` atom. For circular gradients, this specifies the top left offset from the path being drawn. For linear gradients, it is the offset from the center of the rotated gradient line.

For more information about gradients, see “Gradients for Path Fills” (page 848).

Listing 24-6 shows how to create a rectangle with a left-to-right linear gradient fill.

Listing 24-6 Adding a left-to-right linear gradient fill to a rectangle

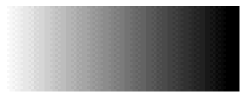
```
ComponentInstance vectorCodec;
Handle streamH;
Handle pathH;
gxPoint points[4];
Boolean isOnCurve[4];
GradientColorRecord gradientColors[2];
ARGBColor white, black;
int i;
long value;
```

```
/* specify the gradient colors and where they start and end */
white.alpha = 255;
white.red = 255;
white.green = 255;
white.blue = 255;
black.alpha = 255;
black.red = 0;
black.green = 0;
black.blue = 0;
gradientColors[1].thisColor = white;
gradientColors[1].endingPercentage = ff(0);
gradientColors[2].thisColor = black;
gradientColors[2].endingPercentage = ff(1);

/* open the vector codec component */
OpenADefaultComponent (decompressorComponentType,
                        kVectorCodecType, &vectorCodec);

/* create a new vector data stream */
CurveCreateVectorStream (vectorCodec, streamH);
value=kLinearGradient;

/* add an atom to the vector data stream that specifies to
   use linear gradient fills for subsequent paths */
CurveAddAtomToVectorStream (ci, kCurveGradientTypeAtom, sizeof(long),
                            &value, streamH);

/* add an atom to the vector data stream that specifies to
   draw linear gradient fills from left to right for subsequent paths */
CurveAddAtomToVectorStream (ci, kCurveGradientAngleAtom, sizeof(Fixed),
                            ff(270), streamH);

/* add an atom to the vector data stream that specifies
   gradient colors */
CurveAddAtomToVectorStream (ci, kCurveGradientRecordAtom,
                            sizeof(GradientColorRecord)*2,
                            gradientColors, streamH);

/* specify the points for the path and whether each one is
   on the path */
```

```
points[0].x = ff(150);
points[0].y = ff(150);
isOnCurve[0] = TRUE;
points[1].x = ff(300);
points[1].y = ff(150);
isOnCurve[1] = TRUE;
points[2].x = ff(300);
points[2].y = ff(100);
isOnCurve[2] = TRUE;
points[3].x = ff(100);
points[3].y = ff(100);
isOnCurve[3] = TRUE;

/* create the path and add the points to it */
CurveNewPath (vectorCodec, &pathH);
for (i = 0; i <= 3; i++)
    CurveInsertPointIntoPath (vectorCodec, &points[i], pathH,
                              1, i, isOnCurve[i]);

/* add the path to the vector data stream */
CurveAddPathAtomToVectorStream (vectorCodec, pathH, streamH);

/* mark the end of the vector data stream by adding a
   kCurveEndAtom atom to the stream */
CurveAddZeroAtomToVectorStream (vectorCodec, streamH);

/* use the vector codec here to decompress and display the vector data */

/* dispose of stream and path handles when done */
DisposeHandle (streamH);
DisposeHandle (pathH);
```

Figure 24-67 shows what is drawn for this vector data stream.

Figure 24-67 A rectangle with a linear left-to-right gradient fill



Specifying a Color for a Path

You specify a color with which to draw paths by including a kCurveARGBColorAtom atom in a vector data stream. The default color is black. This atom is described in “Vector QT Atom Container” (page 874).

As with other atoms that specify path attributes, an attribute value specified by a kCurveTransferModeAtom atom is used for subsequent paths in the vector data stream until another atom of the same type appears in the stream or the end of the stream is reached.

Specifying a Transfer Mode

You specify a transfer mode by including a kCurveTransferModeAtom atom, described in “Vector QT Atom Container” (page 874), in a vector data stream. By default, no transfer mode is used. For descriptions of transfer modes, see “Transfer Modes” (page 814). The data for a kCurveTransferModeAtom atom is a gxTransferMode structure, which is described in “The Transfer Mode Structure” (page 884).

Hit-Testing a Path

You can use the vector codec function CurveGetNearestPathPoint to determine how close a point is to a path. This is useful for hit-testing: you specify a point in a human-interface that a user has clicked, and function returns how far it is from the path. The CurveGetNearestPathPoint function is described in “CurveGetNearestPathPoint” (page 900).

Drawing Vectors

To draw the contents of a vector data stream, you implicitly invoke the vector codec component in the same way that you invoke other QuickTime decompressor components. For information about and examples of decompressing images, see the Chapter 3 of *Inside Macintosh: QuickTime.*

Converting QuickDraw GX Data to QuickTime Vectors

The GX-to-vector transcoder included with QuickTime lets you convert QuickDraw GX data into equivalent QuickTime vectors. If your application already has QuickDraw GX data, or if you use a drawing program that can create QuickDraw GX data, the transcoder makes it easy to create vector graphics for your application.

Note
To use the GX-to-vector transcoder, either QuickDraw GX or the GX Graphics extension must be installed on the computer. ◆

The GX-to-vector transcoder converts all QuickDraw GX data, including all fill, ink, and stroke options, into QuickTime vectors that can be processed by the QuickTime vector codec. The transcoder converts bitmaps, which are not supported by QuickTime, into rectangles. If the GX data includes transfer modes, the minimum bit depth of the resulting vector output is 16.

Listing 24-7 shows how to use the transcoder to convert a QuickDraw GX shape into a handle containing a QuickTime vector data stream.

Listing 24-7 Converting QuickDraw GX data to QuickTime vectors

```
static Handle GXShapeToCurveData(Handle shapeData)
                                    // input: shape data, destroyed
{
    ImageDescriptionHandle      pathDesc = nil;
    ImageTranscodeSequence      ts = nil;
    OSErr           err;
    ImageDescriptionHandleidh = (ImageDescriptionHandle)NewHandle(0);

    // transcode this frame into a path
    err = ImageTranscodeSequenceBegin(&ts, idh, ‘path’, &pathDesc, nil, 
0);
    if (err == noErr)
        {
        void *pathData;
        long pathDataSize;

        HLock(shapeData);
        err = ImageTranscodeFrame(ts, *shapeData,
GetHandleSize(shapeData),
            &pathData, &pathDataSize);
        HUnlock(shapeData);
        if (err == noErr)
            {
            SetHandleSize(shapeData, pathDataSize);
            if (MemError() == noErr)
                {
                BlockMoveData(pathData, *shapeData, pathDataSize);
                DisposeHandle((Handle)idh);
                idh = pathDesc;
                pathDesc = nil;
                }
            ImageTranscodeDisposeFrameData(ts, pathData);
            }

        ImageTranscodeSequenceEnd(ts);
        }

    return(shapeData);

} // GXShapeToCurveData
```

QuickTime Vector Reference

This section describes the constants, data structures, and functions that are specific to QuickTime vectors.

Constants

This section provides details on constants used for QuickTime vectors.

Vector Atom Types

These constants are described in "Vector QT Atom Container" (page 874).

```
/* vector atom types */
enum {
    kCurvePathAtom                      = FOUR_CHAR_CODE('path'),
    kCurveEndAtom                       = FOUR_CHAR_CODE('zero'),
    kCurveAntialiasControlAtom          = FOUR_CHAR_CODE('anti'),
    kCurveFillTypeAtom                  = FOUR_CHAR_CODE('fill'),
    kCurvePenThicknessAtom              = FOUR_CHAR_CODE('pent'),
    kCurveMiterLimitAtom                = FOUR_CHAR_CODE('mitr'),
    kCurveJoinAttributesAtom            = FOUR_CHAR_CODE('join'),
    kCurveMinimumDepthAtom              = FOUR_CHAR_CODE('mind'),
    kCurveTransferModeAtom              = FOUR_CHAR_CODE('xfer'),
    kCurveGradientAngleAtom             = FOUR_CHAR_CODE('angl'),
    kCurveGradientRadiusAtom            = FOUR_CHAR_CODE('radi'),
    kCurveGradientOffsetAtom            = FOUR_CHAR_CODE('cent'),
    kCurveGradientRecordAtom            = FOUR_CHAR_CODE('grad'),
    kCurveARGBColorAtom                 = FOUR_CHAR_CODE('argb'),
    kCurveGradientTypeAtom              = FOUR_CHAR_CODE('grdt')
};
```

Gradient Types

The following constants specify the type of a gradient. Gradients are described in "Gradients for Path Fills" (page 848).

```
enum {
    kLinearGradient         = 0,
    kCircularGradient       = 1
};
```

Constant descriptions

`kLinearGradient`
: Specifies a linear gradient.

`kCircularGradient`
: Specifies a circular gradient.

Fill Types

The following constants specify fill types for paths. For more information about fill types, see “Shape Fill” (page 764).

```
/* fill types */
enum {
    gxNoFill                = 0,
    gxOpenFrameFill         = 1,
    gxFrameFill             = gxOpenFrameFill,
    gxClosedFrameFill       = 2,
    gxHollowFill            = gxClosedFrameFill,
    gxEvenOddFill           = 3,
    gxSolidFill             = gxEvenOddFill,
    gxWindingFill           = 4,
    gxInverseEvenOddFill    = 5,
    gxInverseSolidFill      = gxInverseEvenOddFill,
    gxInverseFill           = gxInverseEvenOddFill,
    gxInverseWindingFill    = 6
};
```

Constant descriptions

`gxNoFill`
The path is not filled.

`gxOpenFrameFill`
The path is framed or stroked along its outline.

`gxFrameFill`
The path is framed or stroked along outline.

`gxClosedFrameFill`
The path is framed, open ends closed off.

`gxHollowFill`
The path is framed, open ends closed off.

`gxEvenOddFill`
The path is filled with even odd rule.

`gxSolidFill`
The path is filled with even odd rule.

`gxWindingFill`
The path is filled using winding rule.

`gxInverseEvenOddFill`
: The area outside of the path is filled.

`gxInverseFill`
: The area outside of the path is filled.

`gxInverseSolidFill`
: The area outside of the path is filled.

`gxInverseWindingFill`
: The area outside of the winding is filled.

Join Constants

These constants specify the type of join for paths. The appearance of sharp and curve joins is shown in “Joins” (page 786).

```
enum {
    gxSharpJoin         = 0x0000,           /* pointed elbow */
    gxCurveJoin         = 0x0001,           /* curved elbow */
    gxLevelJoin         = 0x0002
};
```

`gxSharpJoin`
: Specifies a join with a pointed elbow.

`gxCurveJoin`
: Specifies a join with a curved elbow.

`gxLevelJoin`
: Specifies no join, which results in a cut-off elbow. This is the same as a `gxSharpJoin` join with a miter limit of `0`.

Selectors for Vector Codec Component Functions

The following constants are the selectors for functions of the vector codec component. For descriptions of these functions, see “Vector Codec Component Functions” (page 895).

```
/* selectors for component functions */
enum {
    kCurveGetLengthSelect                         = 0x0100,
    kCurveLengthToPointSelect                     = 0x0101,
    kCurveNewPathSelect                           = 0x0102,
    kCurveCountPointsInPathSelect              = 0x0103,
    kCurveGetPathPointSelect                   = 0x0104,
    kCurveInsertPointIntoPathSelect            = 0x0105,
    kCurveSetPathPointSelect                   = 0x0106,
    kCurveGetNearestPathPointSelect            = 0x0107,
    kCurvePathPointToLengthSelect              = 0x0108,
    kCurveCreateVectorStreamSelect             = 0x0109,
    kCurveAddAtomToVectorStreamSelect          = 0x010A,
    kCurveAddPathAtomToVectorStreamSelect      = 0x010B,
    kCurveAddZeroAtomToVectorStreamSelect      = 0x010C,
    kCurveGetAtomDataFromVectorStreamSelect    = 0x010D
};
```

Data Types

Vector QT Atom Container

A vector QT atom container specifies the characteristics of a QuickTime vector. The container contains atoms for paths and can also contain atoms that specify attributes of the paths, such as their color. For more information about QT atom containers, see “QuickTime Atoms” (page 47).

To render paths, QuickTime traverses the atoms in the container in order. When it finds an atom for a path, it draws the path using the current attribute values. When it finds an atom for an attribute, the new attribute value replaces the previous value of the attribute. If QuickTime has not found an atom for a particular attribute before drawing a path, it uses the default value for the attribute.

All values in a vector QT atom container must be aligned on long-word boundaries and stored in big endian format.

A vector QT atom container can contain any of the following atoms. Atoms that are required are noted, as are atoms that must occur in a particular position within the container.

- `kCurveAntialiasControlAtom`

  Specifies whether subsequent paths in the vector data stream are drawn using antialiasing. Antialiasing is a technique in which edges of the path that do not fall on pixel boundaries are drawn by alpha blending the color of the path with the color of the background. The alpha values take into account errors introduced by the pixelization of the shape and give edges a smoother appearance. Antialiasing slightly slows the drawing of pixels, and it may be desirable to disable it when maximum drawing speed is required.

Legal values for this atom are

- `0` Enables antialiasing
- `0xFFFFFFFF` Disables antialiasing

The data type of the value is `long`, and the size of the atom is `sizeof(long)`.

The default value is `0xFFFFFFFF`.

This atom is optional.

- `kCurveARGBColorAtom`

  Specifies the color for subsequent paths in the vector data stream. Each of the color values (alpha, red, green, and blue) is an unsigned short integer. For most applications, the alpha value should be set to `0xFFFF`.

  The data type of the value is `ARGBColor`, and the size of the atom is `sizeof(ARGBColor)`.

  The default value specifies black with an alpha value of `0xFFFF`.

  This atom is optional.

- `kCurveEndAtom`

  Specifies the end of the vector data stream.

  The size of the atom is `0`.

  This atom is required, and it must be at the end of the vector data stream.

- `kCurveFillTypeAtom`

  Specifies the fill type for subsequent paths in the vector data stream. For more information about fill types, see “Shape Fill” (page 764).

  Legal values for this atom are

  - `gxNoFill` = 0 Nothing is drawn
  - `gxOpenFrameFill` Shape is framed or stroked along outline.
  - `gxFrameFill` Shape is framed or stroked along outline.
  - `gxClosedFrameFill` Shape is framed, open ends closed off.
  - `gxHollowFill` Shape is framed, open ends closed off.
  - `gxEvenOddFill` Shape is filled with even odd rule.
  - `gxSolidFill` Shape is filled with even odd rule.
  - `gxWindingFill` Shape is filled using winding rule.

- □ `gxInverseEvenOddFill` Area outside of shape is filled.
- □ `gxInverseSolidFill` Area outside of shape is filled.
- □ `gxInverseFill` Area outside of shape is filled.
- □ `gxInverseWindingFill` Area outside of winding is filled.

The data type of this atom is `long`, and the size of the atom is `sizeof(long)`.

The default value is `gxEvenOddFill`.

This atom is optional.

- `kCurveGradientAngleAtom`

  Specifies the gradient angle for subsequent linear gradients in the QT atom container. This allows the use of a linear gradient that begins at any part of the path rather than just at the top. The value defines the angle, in degrees, to rotate the linear gradient. For example, 0.0 specifies a top-to-bottom gradient, 90.0 specifies a right-to-left gradient, 180.0 specifies a bottom-to-top gradient, and 270.0 specifies a left-to-right gradient.

  The data type of this atom is `Fixed`, and the size of the atom is `sizeof(Fixed)`.

  The default value is `0.0`.

  This atom is optional.

- `kCurveGradientOffsetAtom`

  Specifies the offset for subsequent gradients in the QT atom container. For circular gradients, this specifies the top left offset from the path being drawn. For linear gradients, it is the offset from the center of the rotated gradient line.

  The data type of this atom is `gxPoint`, and size of this atom is `sizeof(gxPoint)`.

  The default value is no offset.

  This atom is optional.

- `kCurveGradientRadiusAtom`

  Specifies the radius, in pixels, of subsequent circular gradients in the QT atom container. This controls the positioning of the outside color in a circular gradient, which you can use to make a large section of the path be that outermost color.

  The data type of this atom is `Fixed`, and the size of the atom is `sizeof(Fixed)`.

  The default value is `0`, which results in no circular gradient being drawn.

  This atom is required to draw a circular gradient. The value has no effect on a linear gradient.

- `kCurveGradientRecordAtom`

  Specifies the gradient fill colors for subsequent gradients in the QT atom container or, if there is no data, specifies to disable gradient fills for subsequent paths. If there is data, it consists of two or more `GradientColorRecord` structures that specify gradient fill colors. Each of these structures contains the following fields:

  - `thisColor` An `ARGBColor` value that specifies a color.
  - `endingPercentage` A `Fixed` value that specifies the position in the gradient, expressed as a value between `0.0` and `1.0`, at which to use the color. The value `0.0` specifies the beginning of the gradient, and the value `1.0` specifies the end.

  Two `GradientColorRecord` structures are required in `kCurveGradientRecordAtom` atoms that contain data: one for `0.0`, which specifies the top or outside color of the gradient, and one for `1.0`, which specifies the bottom or centermost color used in the gradient. The data can include any number of additional `GradientColorRecord` structures. When QuickTime fills paths using the gradient, it uses a continuum of colors between the specified values.

  The data type of this atom is `GradientColorRecord`, and the size of the atom is `sizeof(GradientColorRecord)*count`, where `count` is the number of `GradientColorRecord` structures in the atom's data.

  The default is no data, which specifies to disable gradient fills for subsequent paths.

  This atom is optional.

- `kCurveGradientTypeAtom`

  Specifies the type of subsequent gradients in the QT atom container. Legal values are:

  - `kLinearGradient` Specifies a linear gradient.
  - `kCircularGradient` Specifies a circular gradient.

  By default, a linear gradient is filled from the top of the path to the bottom in a continuous line of colors. A circular gradient is drawn with a series of overlapping circles to form a burst pattern.

  The data type of this atom is `long`, and the size of the atom is `sizeof(long)`.

  The default value is `kLinearGradient`, which specifies a linear gradient.

  This atom is optional.

- `kCurveJoinAttributesAtom`

Specifies the type of join for subsequent paths in the vector data stream with sharp angles, as described in "Joins" (page 786). When two lines in a polygonal path meet, there are several ways of resolving the intersection of their edge. Legal values are:

- `gxSharpJoin` Specifies a pointed elbow.
- `gxCurveJoin` Specifies a curved elbow.
- `gxLevelJoin` Specifies a cut-off elbow (no join).

The data type of this atom is `gxJoinAttribute`, and the size of the atom is `sizeof(gxJoinAttribute)`.

The default value is `gxSharpJoin`.

This atom is optional.

- `kCurveMinimumDepthAtom`

  Specifies the minimum bit depth for drawing subsequent paths in the vector data stream. If the bit depth of the display device is less than the bit depth specified by the `kCurveMinimumDepthAtom` atom, the path is rendered into an offscreen buffer with the minimum bit depth and then transferred to the screen. This value must be at least 16 for most transfer modes. If the quality of a dithered image is unsatisfactory, increasing the `kCurveMinimumDepthAtom` value to 32 may improve the result.

  The data type of this atom is `long`, and the size of the atom is `sizeof(long)`.

  There is no default value; by default, the bit depth of the output is not changed.

  The `kCurveMinimumDepthAtom` atom is optional.

  If the `kCurveMinimumDepthAtom` atom is included, it must be first atom in the QT atom container.

- `kCurveMiterLimitAtom`

  Specifies the miter limit for subsequent paths in the vector data stream that include a sharp angle, such a polygons and triangles, as described in "Joins" (page 786). The value of this atom is multiplied by the current pen thickness (this is the value of the most recent `kCurvePenThicknessAtom` atom in the vector data stream or, if there is no `kCurvePenThicknessAtom` atom in the stream, the default value `1.0`). The resulting value specifies the maximum number of points (1/72 of an inch) away from the ends at which the join can be drawn. The constant `gxPositiveInfinity` specifies that the join should be as sharp as necessary.

  This value applies only to framed paths (those for which the `kCurveFillTypeAtom` value is `gxOpenFrameFill` or `gxClosedFrameFill`).

The data type of this atom is `Fixed`, and the size of the atom is `sizeof(Fixed)`.

The default value is `1`.

This atom is optional.

- `kCurvePathAtom`

  Specifies a path to draw. The data for the path in contained in a `gxPaths` structure, which can contain up to 32767 contours, each of which may contain up to 32767 points. Each point may be located on or off the curve, with the curve describing a quadratic Bezier. The vectors in the contours are drawn together, obeying the intersection rules for the fill type, if any.

  The data type of this atom is `gxPaths`, and the size of the atom is `sizeof(gxPaths) + n`, where `n` is `(`*`number of points`*`) * sizeof(gxPoint)`.

  This atom is required.

- `kCurvePenThicknessAtom`

  For subsequent paths in the vector data stream that are framed, specifies the thickness of the stroke for the frame, in points (1/72 of an inch). The center-frame attribute is set for the stroke, as described in “Pen Placement” (page 784).

  This value applies only to framed paths (those for which the `kCurveFillTypeAtom` value is `gxOpenFrameFill` or `gxClosedFrameFill`).

  `0` is an illegal value for this atom.

  The data type of this atom is `Fixed`, and the size of the atom is `sizeof(Fixed)`.

  The default value is `1.0`.

  This atom is optional.

- `kCurveTransferModeAtom`

  Specifies the transfer mode for drawing paths. For information about transfer modes, see “Transfer Modes” (page 814).

  The data type of this atom is `gxTransferMode`, and the size of the atom is `sizeof(gxTransferMode)`.

  The default is to draw paths without a transfer mode.

  This atom is optional.

ARGB Color

The `ARGBColor` structure specifies a color with alpha, red, green, and blue values.

```
struct ARGBColor {
    unsigned short      alpha;
    unsigned short      red;
    unsigned short      green;
    unsigned short      blue;
};
typedef struct ARGBColor ARGBColor;
```

Field descriptions

`alpha`	Specifies the alpha component of the color.
`red`	Specifies the red component of the color.
`green`	Specifies the green component of the color.
`blue`	Specifies the blue component of the color.

Gradient Color Record

The `GradientColorRecord` structure specifies a color used in a gradient. For more information about gradients, see “Gradients for Path Fills” (page 848).

```
struct GradientColorRecord {
    ARGBColor               thisColor;
    Fixed                   endingPercentage;
};
typedef struct GradientColorRecord GradientColorRecord;
typedef GradientColorRecord *GradientColorPtr;
```

Field descriptions

`thisColor`	Specifies a color used for a gradient.
`endingPercentage`	Specifies the percentage of the gradient (expressed as value between `0` and `1`, where `0` is the beginning of the gradient) at which the specified color begins.

The Point Structure

The `gxPoint` structure is defined as follows:

```
struct gxPoint {
    Fixed   x;
    Fixed   y;
};
```

Field descriptions

x	A horizontal distance. Greater values of the x field indicate distances further to the right.
y	A vertical distance. Greater values of the y field indicate distances further down.

The location of the origin depends on the context where you use the point; for example, it might be the upper-left corner of a view port.

Notice that the x and y fields are of type `Fixed`. QuickDraw GX allows you to specify fractional coordinate positions.

The Path Structure

You use the `gxPath` structure to specify a single contour composed of straight lines and curves. For information about paths, see “Path Shapes” (page 771).

The `gxPath` structure is defined as follows:

```
struct gxPath {
    long                vectors;
    long                controlBits[gxAnyNumber];
    struct gxPoint      vector[gxAnyNumber];
};
```

Field descriptions

vectors	The number of geometric points in the contour.
controlBits	Bit flags that indicate which geometric points are on curve and which are off-curve control points.
vector	The coordinates of the geometric points.

The array index `gxAnyNumber` indicates that the `gxPath` data structure is a variable-length structure—it can include any number of geometric points

Each bit in the array specified in the `controlBits` field indicates whether a particular point in the array specified by the vector field is on curve or off curve. A value of 0 indicates that the corresponding point is on curve and a value of 1 indicates that the corresponding point is off curve.

The Paths Structure

The `gxPaths` structure allows you to group multiple path contours together. You use this data structure when specifying the geometry of a path shape. For information about paths, see "Path Shapes" (page 771).

The `gxPaths` structure is defined as follows:

```
struct gxPaths {
    long            contours;
    struct gxPath   contour[gxAnyNumber];
};
```

Field descriptions

`contours`	The number of path contours.
`contour`	The path contours.

The array index `gxAnyNumber` indicates that the `gxPaths` data structure is also a variable-length structure—it can include any number of path contours.

Note
In QuickTime 3, a single path contour can have between 0 and 32,767 geometric points. The geometry of a path shape can between 0 and 32,767 polygon contours. The total size of a path geometry may not exceed 2,147,483,647 bytes. ◆

For more information about paths, see "Path Shapes" (page 771) and "Creating and Drawing Paths" (page 774).

Color Structure

A color value, plus a specification of the color space it belongs to, constitute a color. A color is a structure defined by the `gxColor` type definition:

```
struct gxColor{
    gxColorSpace            space;
    gxColorProfile          profile;
    union {
            struct gxCMYKColor            cmyk;
            struct gxRGBColor             rgb;
            struct gxRGBAColor            rgba;
            struct gxHSVColor             hsv;
            struct gxHLSColor             hls;
            struct gxXYZColor             xyz;
            struct gxYXYColor             yxy;
            struct gxLUVColor             luv;
            struct gxLABColor             lab;
            struct gxYIQColor             yiq;
            gxColorValue                  gray;
            struct gxGrayAColor           graya;
            unsigned short                pixel16;
            unsigned long                 pixel32;
            struct gxIndexedColor         indexed;
            gxColorValue                  component[4];
    } element;
};
```

Field descriptions

`space`	The color space for this color.
`profile`	For QuickTime vectors, this must be `nil`.
`element`	The color value for this color.

The `element` field is a union that can contain any one of the following fields:

`cmyk`	A CMYK color value.
`rgb`	An RGB color value.
`rgba`	An alpha-channel RGB color value.
`hsv`	An HSV color value.
`hls`	An HLS color value.
`xyz`	An XYZ color value.
`yxy`	A Yxy color value.
`luv`	An L*u*v* color value.
`lab`	An L*a*b* color value.
`yiq`	A YIQ color value.
`gray`	A grayscale color value
`graya`	An alpha-channel grayscale color value.
`pixel16`	A 16-bit pixel value, in `gxRGB16Space` format.

`pixel32`	A 32-bit pixel value, in any of the 32-bit color space formats.
`indexed`	Unused for QuickTime vectors.
`component`	An array of 4 undefined color-component values. Useful for indexing through the color one component at a time, as when working with different transfer modes for each color component.

The Transfer Mode Structure

The `gxTransferMode` structure specifies the transfer mode property of an ink object. For information about transfer modes, see “Transfer Modes” (page 814).

The structure is defined as follows:

```
struct gxTransferMode {
    gxColorSpace        space;      /* the gxColor-space the transfer
                                       mode is to operate in */
    gxColorSet          set;
    gxColorProfile      profile;
    Fixed               sourceMatrix[5][4];
    Fixed               deviceMatrix[5][4];
    Fixed               resultMatrix[5][4];
    gxTransferFlag      flags;
    gxTransferComponent component[4];  /* how each component is
                                          operated upon */
};
typedef struct gxTransferMode gxTransferMode;
```

Field descriptions

`space`	The color space used by this transfer mode. The color spaces defined by QuickDraw GX are listed in the color structure definition shown in the previous section, and are described in “Color Spaces” (page 791). A transfer mode’s color space need not be the same as the color space of the ink object the transfer mode is part of.
`set`	For QuickTime vectors, this field must be `nil`.
`profile`	For QuickTime vectors, this field must be `nil`.

`sourceMatrix`	A 5 x 4 matrix that specifies how to transform the source color before applying the component modes to the components. See “Transfer Mode Matrices” (page 840) for more information.
`deviceMatrix`	A 5 x 4 matrix that specifies how to transform the destination color before applying the component modes to the components. See “Transfer Mode Matrices” (page 840) for more information.
`resultMatrix`	A 5 x 4 matrix that specifies how to transform the result color after applying the component modes to all components. See “Transfer Mode Matrices” (page 840) for more information.
`flags`	The transfer mode flags; they specify how to handle color limits and whether multiple transfer components are needed. The transfer mode flags are described in “Transfer Mode Flags” (page 843).
`component`	An array of four transfer component structures, each specifying the type of transfer mode to apply to a single color component of the transfer mode’s color space. For a description of the transfer component structure, see “Transfer Component Structure” (page 885).

Transfer Component Structure

There are up to four transfer component structures in a transfer mode structure. Each transfer component structure describes how the transfer mode operation is to be applied to a given component in the transfer mode’s color space. For information about transfer modes, see “Transfer Modes” (page 814).

The transfer component structure is of data type `gxTransferComponent`:

```
struct gxTransferComponent{
        gxComponentMode         mode;
        gxComponentFlag         flags;
        gxColorValue            sourceMinimum;
        gxColorValue            sourceMaximum;
        gxColorValue            deviceMinimum;
        gxColorValue            deviceMaximum;
        gxColorValue            clampMinimum;
        gxColorValue            clampMaximum;
        gxColorValue            operand;
};
```

`mode`	The component mode (type of transfer mode, such as `gxCopyMode` or `gxBlendMode`) to apply to this color component. Component modes are described in the section “Transfer Mode Types” (page 814).
`flags`	The component flags, which control clamping behavior and whether source and destination are to be reversed for this component. The component flags are described in the section “Transfer Component Flags” (page 842).
`sourceMinimum`	The minimum acceptable value for source color in this color component. No drawing occurs if the source component value is below `sourceMinimum`. For more information, see “Color Limits” (page 834), and “Source Color Limits” (page 838).
`sourceMaximum`	The maximum acceptable value for source color in this color component. No drawing occurs if the source component value is greater than `sourceMaximum`. For more information, see “Color Limits” (page 834), and “Source Color Limits” (page 838).
`deviceMinimum`	The minimum acceptable value for destination color in this color component. No drawing occurs if the destination component value is below `deviceMinimum`. For more information, see “Color Limits” (page 834), and “Destination Color Limits” (page 839).
`deviceMaximum`	The maximum acceptable value for destination color in this color component. No drawing occurs if the destination component value is greater than `deviceMaximum`. For more information, see “Color Limits” (page 834), and “Destination Color Limits” (page 839).
`clampMinimum`	The minimum acceptable value for result color in this color component. If the result component value is below `deviceMinimum`, it is pinned, or clamped, to the value of `deviceMinimum`. For more information, see “Color Limits” (page 834), and “Result Color Limits” (page 839).
`clampMaximum`	The maximum acceptable value for result color in this color component. If the result component value is greater

	than deviceMaximum, it is pinned, or clamped, to the value of deviceMaximum. For more information, see “Color Limits” (page 834), and “Result Color Limits” (page 839).
operand	A value used as an input to the gxBlendMode, gxMigrateMode, and gxHighlightMode component modes. If you are using these modes, you must supply a proper value for operand. For more information, see “Arithmetic Transfer Modes” (page 815), and “Highlight Transfer Mode” (page 819).

Transfer Component Flag Type

The gxComponentFlag type is used for transfer component flags.

```
typedef unsigned char   gxComponentFlag;
```

These flags, which are in the in the component field of the gxTransferComponent structure, alter the source, destination, or result value for an individual color component. For more information about the transfer modes for QuickTime vectors, see “Transfer Modes” (page 814).

The following transfer component flags are defined:

Constant	Value	Explanation
gxOverResultComponent	0x01	QuickDraw GX performs an AND operation between the result color and 0xFFFF before clamping.
gxReverseComponent	0x02	QuickDraw GX reverses the source and destination values before performing the transfer mode operation.

Specifying gxOverResultComponent allows the result of transfers using gxAddMode to wrap around (from 0xFFFF to 0x0000) instead of remaining clamped at 0xFFFF.

Specifying gxReverseComponent allows you to apply a transfer mode backwards—from the destination to the source—for a particular component. It is most useful for component modes that depend on order, like gxMigrateMode, or gxAddMode when used for subtraction.

Transfer Component Mode Type

The `gxComponentMode` type is used for transfer component mode constants.

```
typedef unsigned char   gxComponentMode;
```

A transfer component mode constants specifies the type of transfer mode (like "copy" or "XOR") to use. For more information about the transfer modes for QuickTime vectors, see "Transfer Modes" (page 814).

The following transfer mode constants are defined:

Constant	Value	Explanation
`gxNoMode`	0	No mode. No transfer occurs. For this component of the color, the destination is left as it was. This mode is useful for suppressing drawing when certain logical conditions are met, or for not drawing one color component while allowing other components to be drawn.
`gxCopyMode`	1	Copy mode. The source color component is copied to the destination. The destination component is ignored. This is the most common transfer mode, and is the default for QuickDraw GX.
`gxAddMode`	2	Add mode. The source color component is added to the destination component, but the result is not allowed to exceed the maximum value (`0xFFFF` or `gxColorValue1`; white in Figure 24-39).
`gxBlendMode`	3	Blend mode. The result is the average of the source and destination color components, weighted by a ratio specified by the operand component (0.5 in Figure 24-39). The operand varies from 0 (all destination) to `0xFFFF` or `gxColorValue1` (all source), although it is customary to interpret it as varying between `0` and `1`.

Constant	Value	Explanation
`gxMigrateMode`	4	Migrate mode. The destination color component is moved toward the source component by the value of the step specified in the operand component (0.25, or `0x4000` in Figure 24-39 (page 816)). Migrate mode is similar to blend mode, except that the change in destination component is an absolute amount, rather than a proportion of the difference between it and the source component. If the source has a greater color component value than the destination, the migration is positive; if the destination has a greater value than the source, the migration is negative. In either case, the amount of migration cannot be greater than the difference between the destination and the source values.
`gxMinimumMode`	5	Minimum mode. The source component replaces the destination component only if the source component has a smaller value. (In Figure 24-39 (page 816), drawing occurs only within the area occupied by the cloud.)
`gxMaximumMode`	6	Maximum mode. The source component replaces the destination component only if the source component has a larger value. (In Figure 24-39 (page 816), drawing occurs only outside of the area occupied by the cloud.)
`gxHighlightMode`	7	Highlight mode. The source component and operand component are swapped in the destination. Other components in the destination are ignored.
`gxAndMode`	8	AND mode. The bits of the source color and destination color are combined using an AND operation. Only bits that are set in both source and destination remain set in the result.
`gxOrMode`	9	OR mode. The bits of the source color and destination color are combined using an OR operation. Bits that are set in either the source or the destination or in both are set in the result.

Constant	Value	Explanation
`gxXorMode`	10	XOR mode. The bits of the source color and destination color are combined using an exclusive-OR (XOR) operation. Bits that are set in the source but not the destination, and bits that are set in the destination but not the source, are set in the result. All other bits are cleared in the result.
`gxRampAndMode`	11	Ramp-AND mode. The source and destination color components are treated as ranging from 0 to 1; their product (source × destination) is returned.
`gxRampOrMode`	12	Ramp-OR mode. The source and destination color components are treated as ranging from 0 to 1; the result of (source + destination – source × destination) is returned.
`gxRampXorMode`	13	Ramp-XOR mode. The source and destination color components are treated as ranging from 0 to 1; the result of (source + destination – 2 × source × destination) is returned.
`gxOverMode`	14	Over mode. The source color is copied to the destination, and the source transparency controls where the destination color shows through. Where both are transparent, no drawing occurs (result equals destination).

Constant	Value	Explanation
`gxAtopMode`	15	Atop mode. The source color is placed over the destination, but the resulting destination retains the original destination's transparency. The effect is that opaque parts of the source are clipped to cover only opaque parts of the destination.
`gxExcludeMode`	16	Exclude mode. The destination color remains visible only where the source is transparent, and the source color is copied anywhere the destination is transparent. Where both are transparent, no drawing occurs (result equals destination); where both are opaque, the result color is 0 (no intensity).
`gxFadeMode`	17	Fade mode. The source is blended with the destination, using the relative alpha values as the ratio for the blend. Where both are transparent, the result is the average of the source and the destination).

Color Value Type

The `gxColorValue` type is used for a value of one of a color's components, such as red in an RGB color specification.

```
typedef unsigned short  gxColorValue;
```

For more information about color-component values, see "Color-Component Values, Color Values, and Colors" (page 812).

Transfer Mode Flag Type

The `gxTransferFlag` type is used for a transfer-mode flag.

```
typedef long            gxTransferFlag;
```

The transfer mode flags are a set of flags in the `flags` field of the transfer mode structure. They affect how color limits are used and whether a single component mode is to be used for all color components. For more information about transfer modes, see “Transfer Modes” (page 814).

The following three flags are defined.

Constant	Value	Explanation
`gxRejectSourceTransfer`	0x0001	Negate the results of `sourceMinimum` and `sourceMaximum` for all components. Accept only values *outside* of the specified ranges.
`gxRejectDeviceTransfer`	0x0002	Negate the results of `deviceMinimum` and `deviceMaximum` for all components. Accept only values *outside* of the specified ranges.
`gxSingleComponentTransfer`	0x0004	Use a single transfer component for all color components. Duplicate `component[0]` in the transfer mode structure for all components in the transfer mode’s color space.

Setting the `gxRejectSourceTransfer` or `gxRejectDeviceTransfer` flag causes an inversion of the acceptable color ranges for source or destination color, respectively. For example, in Figure 24-51 (page 837), setting the `gxRejectSourceTransfer` or `gxRejectDeviceTransfer` flag would cause the white (empty) portions of the large cubes that represent RGB space to be within range, instead of the gray (filled) portions.

The effect is similar to, although not exactly the same as, individually reversing the minimum and maximum values for the color components. If the transfer mode flag is cleared, drawing occurs only when *all* components are inside the allowed ranges—that is, inside the darker gray portions of the large cubes in Figure 24-51. With the flag set, drawing occurs any time *at least one* component is outside of its allowed range—that is, with values anywhere outside of the dark gray areas in Figure 24-51.

The `gxSingleComponentTransfer` flag is provided as a convenience. You can set the flag when you don’t need the flexibility (and extra effort) of specifying different transfer modes for different color components. In this case you need set up only one `gxTransferComponent` structure, instead of one for each component in the transfer mode’s color space.

Color Space Type

The `gxColorSpace` type is used for a specification of a color space.

```
typedef long            gxColorSpace;
```

For information about color spaces, see “Color Spaces” (page 791).

Color Space Structures

QuickTime uses the following structures to define color spaces. For information about the color spaces supported by QuickTime, see “Color Spaces” (page 791).

```
struct gxRGBColor {
    gxColorValue        red;
    gxColorValue        green;
    gxColorValue        blue;
};
typedef struct gxRGBColor gxRGBColor;

struct gxRGBAColor {
    gxColorValue        red;
    gxColorValue        green;
    gxColorValue        blue;
    gxColorValue        alpha;
};
typedef struct gxRGBAColor gxRGBAColor;

struct gxHSVColor {
    gxColorValue        hue;
    gxColorValue        saturation;
    gxColorValue        value;
};
typedef struct gxHSVColor gxHSVColor;

struct gxHLSColor {
    gxColorValue        hue;
    gxColorValue        lightness;
    gxColorValue        saturation;
};
typedef struct gxHLSColor gxHLSColor;
```

```
struct gxCMYKColor {
    gxColorValue        cyan;
    gxColorValue        magenta;
    gxColorValue        yellow;
    gxColorValue        black;
};
typedef struct gxCMYKColor gxCMYKColor;

struct gxXYZColor {
    gxColorValue        x;
    gxColorValue        y;
    gxColorValue        z;
};
typedef struct gxXYZColor gxXYZColor;

struct gxYXYColor {
    gxColorValue        capY;
    gxColorValue        x;
    gxColorValue        y;
};
typedef struct gxYXYColor gxYXYColor;

struct gxLUVColor {
    gxColorValue        l;
    gxColorValue        u;
    gxColorValue        v;
};
typedef struct gxLUVColor gxLUVColor;

struct gxLABColor {
    gxColorValue        l;
    gxColorValue        a;
    gxColorValue        b;
};
typedef struct gxLABColor gxLABColor;

struct gxYIQColor {
    gxColorValue        y;
    gxColorValue        i;
    gxColorValue        q;
};
typedef struct gxYIQColor gxYIQColor;

struct gxGrayAColor {
    gxColorValue        gray;
    gxColorValue        alpha;
};
typedef struct gxGrayAColor gxGrayAColor;
```

Vector Codec Component Functions

The following functions are provided by the vector codec component.

CurveAddAtomToVectorStream

You use the `CurveAddAtomToVectorStream` function to add an atom to a vector data stream.

```
pascal ComponentResult CurveAddAtomToVectorStream (
                     ComponentInstance effect,
                     OSType atomType,
                     Size atomSize,
                     void *pAtomData,
                     Handle vectorStream);
```

`effect`	Specifies the instance of the QuickTime vector codec component for the request. Your software obtains this reference when calling the Component Manager's `OpenComponent` or `OpenDefaultComponent` function.
`atomType`	Specifies the type of atom to add to the vector data stream.
`atomSize`	Specifies the size of the data for the atom.
`pAtomData`	Contains a pointer to the data for the atom.
`vectorStream`	Contains a handle to the vector data stream to which to add the atom.

DISCUSSION

This function adds the atom to the end of the specified vector data stream and resizes the vector data stream handle.

CurveAddPathAtomToVectorStream

You use the `CurveAddPathAtomToVectorStream` function to add a path to a vector data stream.

```
pascal ComponentResult CurveAddPathAtomToVectorStream (
                     ComponentInstance effect,
                     Handle pathData,
                     Handle vectorStream);
```

`effect`	Specifies the instance of the QuickTime vector codec component for the request. Your software obtains this reference when calling the Component Manager's `OpenComponent` or `OpenDefaultComponent` function.
`pathData`	Contains a handle to the data for the path.
`vectorStream`	Contains a handle to the vector data stream to which to add the path.

DISCUSSION

This function adds the path to the end of the specified vector data stream and resizes the vector data stream handle.

CurveAddZeroAtomToVectorStream

You use the `CurveAddZeroAtomToVectorStream` function to add a `kCurveEndAtom` to a vector data stream. This atom marks the end of the vector data stream,

```
pascal ComponentResult CurveAddZeroAtomToVectorStream (
                     ComponentInstance effect,
                     Handle vectorStream);
```

`effect`	Specifies the instance of the QuickTime vector codec component for the request. Your software obtains this reference when calling the Component Manager's `OpenComponent` or `OpenDefaultComponent` function.
`vectorStream`	Contains a handle to the vector data stream to which to add the `kCurveEndAtom` atom.

DISCUSSION

This function adds a `kCurveEndAtom` atom to the end of the specified vector data stream and resizes the vector data stream handle. The `kCurveEndAtom` atom is required at the end of a vector data stream, and there may be only one `kCurveEndAtom` atom in the stream.

CurveCountPointsInPath

You use the `CurveCountPointsInPath` function to count the points along either one of a path's contours or all of its contours.

```
pascal ComponentResult CurveCountPointsInPath (
                        ComponentInstance effect,
                        gxPaths *aPath,
                        unsigned long contourIndex,
                        unsigned long *pCount);
```

`effect`	Specifies the instance of the QuickTime vector codec component for the request.
`aPath`	Contains pointer to the path.
`contourIndex`	Contains the index of the contour to be counted.
`pCount`	Contains a pointer to a field that is to receive the number of points in the contour or path.

CurveCreateVectorStream

You use the `CurveCreateVectorStream` function to create a new, empty vector data stream.

```
pascal ComponentResult CurveCreateVectorStream (
                     ComponentInstance effect,
                     Handle *pStream);
```

`effect`	Specifies the instance of the QuickTime vector codec component for the request. Your software obtains this reference when calling the Component Manager's `OpenComponent` or `OpenDefaultComponent` function.
`pStream`	Contains a pointer to the handle that is to receive the newly created vector data stream.

DISCUSSION

The caller is responsible for disposing of the stream when finished with it. This can be done by calling the `DisposeHandle` function.

CurveGetAtomDataFromVectorStream

You use the `CurveGetAtomDataFromVectorStream` function to find the first atom of a specified type within a vector data stream and get its data.

```
pascal ComponentResult CurveGetAtomDataFromVectorStream (
                     ComponentInstance effect,
                     Handle vectorStream,
                     long atomType,
                     long *dataSize,
                     Ptr *dataPtr);
```

`effect`	Specifies the instance of the QuickTime vector codec component for the request. Your software obtains this reference when calling the Component Manager's `OpenComponent` or `OpenDefaultComponent` function.

`vectorStream`	Contains a handle to the vector data stream from which to get the atom.
`atomType`	Specifies the type of atom to find.
`dataSize`	Contains a pointer to a field that is to receive the size of the atom's data.
`dataPtr`	Contains a pointer to a pointer to a field that is to receive the atom's data.

DISCUSSION

Before calling this function, your software must lock the handle for the vector data stream (by calling the `HLock` function). This prevents the handle from being moved, which could invalidate the pointer to the atom data before your software gets the data.

CurveGetLength

You use the `CurveGetLength` function to calculate the length of one of a path's contours or the sum of the lengths of all of its contours.

```
pascal ComponentResult CurveGetLength (
                    ComponentInstance effect,
                    gxPaths *target,
                    long index,
                    wide *wideLength);
```

`effect`	Specifies the instance of the QuickTime vector codec component for the request. Your software obtains this reference when calling the Component Manager's `OpenComponent` or `OpenDefaultComponent` function.
`target`	Contains a pointer to the path.
`index`	Contains the index of the contour whose length is to be calculated or, if the value is `0`, specifies to calculate the lengths of all of the path's contours and return the sum of the lengths.
`wideLength`	Contains a pointer to a field that is to receive the length.

CurveGetNearestPathPoint

You use the `CurveGetNearestPathPoint` function to find the closest point on a path to a specified point.

```
pascal ComponentResult CurveGetNearestPathPoint (
                    ComponentInstance effect,
                    gxPaths *aPath,
                    FixedPoint *thePoint,
                    unsigned long *contourIndex,
                    unsigned long *pointIndex,
                    Fixed *theDelta);
```

`effect`	Specifies the instance of the QuickTime vector codec component for the request. Your software obtains this reference when calling the Component Manager's `OpenComponent` or `OpenDefaultComponent` function.
`aPath`	Contains a pointer to the path.
`thePoint`	Contains a pointer to a point for which to find the closest point on the path.
`contourIndex`	Contains a pointer to a field that is to receive the index of the contour that contains the closest point.
`pointIndex`	Contains a pointer to a field that is to receive the index of the closest point.
`theDelta`	Contains a pointer to a field that is to receive the distance between the specified point and the closest point in the contour to it.

DISCUSSION

In programs where users directly manipulate curves, you can use this function to determine the closest control point to a given point.

CurveGetPathPoint

You use the `CurveGetPathPoint` function to get a point from a path and to find out if the point is on the curve.

```
pascal ComponentResult CurveGetPathPoint (
                        ComponentInstance effect,
                        gxPaths *aPath,
                        unsigned long contourIndex,
                        unsigned long pointIndex,
                        gxPoint *thePoint,
                        Boolean *ptIsOnPath);
```

`effect`	Specifies the instance of the QuickTime vector codec component for the request. Your software obtains this reference when calling the Component Manager's `OpenComponent` or `OpenDefaultComponent` function.
`aPath`	Contains a pointer to the path.
`contourIndex`	Specifies the index of the contour from which to get the point.
`pointIndex`	Specifies the index of the point to get.
`thePoint`	Contains a pointer to a field that is to receive the point.
`ptIsOnPath`	Contains a pointer to a field that is to receive a Boolean value that, if `TRUE`, specifies that the point is on the curve.

DISCUSSION

This function lets programs get a single point from a path without walking the path data.

CurveInsertPointIntoPath

You use the `CurveInsertPointIntoPath` function to add a new point to a path.

```
pascal ComponentResult CurveInsertPointIntoPath (
                        ComponentInstance effect,
                        gxPoint *aPoint,
                        Handle thePath,
                        unsigned long contourIndex,
                        unsigned long pointIndex,
                        Boolean ptIsOnPath);
```

`effect`	Specifies the instance of the QuickTime vector codec component for the request. Your software obtains this reference when calling the Component Manager's `OpenComponent` or `OpenDefaultComponent` function.
`aPoint`	Contains a pointer to the point to add to the path.
`thePath`	Contains a handle to the path to which to add the point.
`contourIndex`	Specifies the index of the path contour to which to add the point.
`pointIndex`	Specifies the index of the point to add.
`ptIsOnPath`	If `TRUE`, specifies that the new point is to be on the path.

DISCUSSION

This function is best for adding a single point to a path rather than large numbers of points.

CurveLengthToPoint

You use the `CurveLengthToPoint` function to get the point at a specified distance along a curve.

```
pascal ComponentResult CurveLengthToPoint (
                    ComponentInstance effect,
                    gxPaths *target,
                    long index,
                    Fixed length,
                    FixedPoint *location,
                    FixedPoint *tangent);
```

`effect`	Specifies the instance of the QuickTime vector codec component for the request. Your software obtains this reference when calling the Component Manager's `OpenComponent` or `OpenDefaultComponent` function.
`target`	Contains a pointer to the path.
`index`	Specifies the index of the path contour from which to get the point.
`length`	Specifies the distance along the curve at which to find the point.
`location`	Contains a pointer to a field that is to receive the point.
`tangent`	Contains a pointer to a field that is to receive a point that is tangent to the point at the specified distance.

DISCUSSION

This function is useful for converting a value to a point, such as when creating an animation that follows a curve.

CurveNewPath

You use the `CurveNewPath` function to create a new path.

```
pascal ComponentResult CurveNewPath (
                    ComponentInstance effect,
                    Handle *pPath);
```

`effect`	Specifies the instance of the QuickTime vector codec component for the request. Your software obtains this reference when calling the Component Manager's `OpenComponent` or `OpenDefaultComponent` function.
`pPath`	Contains a pointer to a handle that is to receive the new path.

DISCUSSION

The path created by this function contains one contour and no points. The caller must dispose of the handle when it is finished with it by calling the `DisposeHandle` function.

CurvePathPointToLength

You use the `CurvePathPointToLength` function to get the length of a path between specified starting and ending distances that is nearest a point.

```
pascal ComponentResult CurvePathPointToLength (
                        ComponentInstance ci,
                        gxPaths *aPath,
                        Fixed startDist,
                        Fixed endDist,
                        FixedPoint *thePoint,
                        Fixed *pLength );
```

`ci`	Specifies the instance of the QuickTime vector codec component for the request. Your software obtains this reference when calling the Component Manager's `OpenComponent` or `OpenDefaultComponent` function.
`aPath`	Contains a pointer to the path.
`startDist`	Specifies the distance along the path at which to start measuring the path's length.
`endDist`	Specifies the distance along the path at which to stop measuring the path's length.
`thePoint`	Contains a pointer to a point; the function measures the path closest to this point.
`pLength`	Contains a pointer to a field that is to receive the length of the specified part of the path.

DISCUSSION

You can use this function to test if the user has clicked on the specified portion of the curve.

CurveSetPathPoint

You use the `CurveSetPathPoint` function to change to location of a point in a path.

```
pascal ComponentResult CurveSetPathPoint (
                    ComponentInstance effect,
                    gxPaths *aPath,
                    unsigned long contourIndex,
                    unsigned long pointIndex,
                    gxPoint *thePoint,
                    Boolean ptIsOnPath);
```

`effect`	Specifies the instance of the QuickTime vector codec component for the request. Your software obtains this reference when calling the Component Manager's `OpenComponent` or `OpenDefaultComponent` function.
`aPath`	Contains a pointer to the path.
`contourIndex`	Specifies the index of the path contour that contains the point to change.
`pointIndex`	Specifies the index of the point to change.
`thePoint`	Contains a pointer to the new value for the point.
`ptIsOnPath`	If `TRUE`, specifies that the new point is to be on the path.

DISCUSSION

This function edits an existing point location within a path.

Vector Codec Macro

The `ff` macro is provided by the vector codec component to convert an integer to a `Fixed` value.

```
ff(i);
```

C H A P T E R 2 5

Tween Components and Native Tween Types

This chapter describes the components that perform tween operations, sometimes called **tweeners.** It also describes tween operations performed by QuickTime for tween types that are native to QuickTime.

A tween operation lets you algorithmically generate an output value for any point in a time interval. The input for a tween is a small number of values, often as few as one or two, from which a range of values can be derived. You can use output from a tween to modify tracks in a QuickTime movie, as described in Chapter 13, “Tween Media Handler.” You can also use the output values in your application to perform other actions, unrelated to movies.

This chapter is divided into the following major sections:

- “About Tween Operations” (page 907) introduces tweens and their uses and provides an overview of the tween operations that are possible.
- “Using Tween Components” (page 911) describes how to create the tween containers that the tween media handler uses.
- “Creating a Tween Component” (page 946) explains how to create a tween component for a new data type, interpolation algorithm, or both.
- “Tween Component and Native Tween Type Reference” (page 948) describes the native tween types handled by QuickTime; the tween components included in QuickTime; and the constants, data types, and routines associated with tween components.

About Tween Operations

Every **tween** operation is based on a collection of one or more values from which a range of output values can be algorithmically derived. Each tween is assigned a time duration, and an output value can be generated for any time value within the duration. In the simplest kind of tween operation, a pair of values is provided as input and values be*tween* the two values are generated as output. For example, if the tween data is a pair of integers, 0 and 5, the duration of the tween operation is 100, and the algorithm used to generate output values is linear interpolation (in which generated values, when graphed, fall on a straight line between the input values), the output returned for a time value of 0 is 0, the output for 25 is 1.25, the output for 50 is 2.5, and the output for 100 is 5.

QuickTime supports a variety of **tween types**. Each tween type is distinguished from other types by these characteristics:

- Input values or structures of a particular type.
- A particular number of input values or structures (most often one or two).
- Output values or structures of a particular type.
- A particular algorithm used to derive the output values.

Tween operations for each tween type are performed by a tween component that is specific to that type or, for a number of tween types that are native to QuickTime, by QuickTime itself. Movies and applications that use tweens do not need to specify the tween component to use; QuickTime identifies a tween type by its tween type identifier and automatically routes its data to the correct tween component or to QuickTime. If you need to perform tween operations that QuickTime does not support, you can develop a new tween component, as described in "Creating a Tween Component" (page 946).

When a movie contains a tween track, the tween media handler (described in Chapter 13, "Tween Media Handler") invokes the necessary component (or built-in QuickTime code) for tween operations and delivers the results to another media handler. The receiving media handler can then use the values it receives to modify its playback. For example, the data in a tween track can be used to alter the volume of a sound track.

Tweens can also be used outside of movies by applications or other software that can use the values they generate.

Tween Types

Each of the tween types supported by QuickTime belongs to one of these categories:

- Numeric tween types, which have pairs of numeric values, such as long integers, as input. For these types, linear interpolation is used to generate output values.
- QuickDraw tween types, most of which have pairs of QuickDraw structures, such as points or rectangle, as input. For these types, one or more structure elements are interpolated, such as the `h` and `v` values for points, and each element that is interpolated is interpolated separately from others.
- 3D tween types, which have a QuickDraw 3D structure such as `TQ3Matrix4x4` or `TQ3RotateAboutAxisTransformData` as input. For these types, a specific 3D transformation is performed on the data to generate output.
- The polygon tween type, which takes three four-sided polygons as input. One polygon (such as the bounds for a sprite or track) is transformed, and the two others specify the start and end of the range of polygons into which the tween operation maps it. You can use the output (a `MatrixRecord` data structure) to map the source polygon into any intermediate polygon. The intermediate polygon is interpolated from the start and end polygons for each particular time in the tween duration.
- Path tween types have as input a QuickTime vector data stream for a path. (For information about QuickTime vectors, see Chapter 25, “Tween Components and Native Tween Types.”) Four of the path tween types also have as input a percentage of path’s length; for these types, either a point on the path or a `MatrixRecord` data structure is returned. Two other path tween types treat the path as a function: one returns the `y` value of the point on the path with a given `x` value, and the other returns the `x` value of the point on the path with a given `y` value.
- The list tween type has as input a QT atom container that contains leaf atoms of a specified atom type. For this tween type, the duration of the tween operation is divided by the number of leaf atoms of the specified type. For time points within the first time division, the data for the first leaf atom is returned; for the second time division, the data for the second leaf atom is returned; and so on. The resulting tween operation proceeds in discrete steps (one step for each leaf atom), instead of the relatively continuous tweening produced by other tween types.

For complete definitions of the tween types supported by QuickTime, see “Native Tween Types Processed by QuickTime” (page 949) and “Tween Components for QuickDraw 3D” (page 952).

Single Tweens and Tween Sequences

A tween operation can include one or more tweens. A tween operation that includes just one tween is called a **single tween**. For a single tween, results for any time points in the tween duration are derived from that tween. A **tween sequence** contains more than one tween of the same type. To specify when each tween is used, you specify an ending percentage for each tween. For example, if you have a tween sequence containing three tweens and want to use the first tween for the first quarter of the tween duration, the second tween for the second quarter of the tween duration, and the third tween for the remainder of the tween duration, you set the end percentage for the first tween to `.25`, for the second to `.5`, and for the third to `1.0`. The first tween in the sequence always begins at the beginning, and each subsequent tween begins after the end percentage of the tween before it.

Interpolation Tweens

Interpolation tweens are tweens that modify other tweens. The output of an interpolation tween must be a time value, and the time values generated are used in place of the input time values of the tween being modified. For example, you can use a path tween whose data specifies a curve to modify a tween that uses linear interpolation for its algorithm. The starting and ending values for the modified tween remain the same, but the rate at which output values change over time is determined by the shape of the curve.

Once you create an interpolation tween, you can use it to modify any number of other tweens. You can do this by specifying maximum and/or minimum output values of the interpolation tween to match the time values for the tween to be modified. For example, if there is a curve whose shape describes the natural decay rate for a several different sounds, you can can define a single interpolation tween for that curve and apply it, with appropriate maximum and minimum values, to all of the sounds.

An interpolation tween can modify another interpolation tween; the only requirement is that the output of each interpolation tween must be a time value. The ability to define series of interpolations makes it possible to create libraries of standard modifications that can be used together to generate more complex transformations.

Using Tween Components

The tween media handler uses tween components to perform tween operations (other than simple ones built into QuickTime). The components use containers to define their tween operations, and the containers are constructed from QT atoms. The tween media handler is discussed in Chapter 13, “Tween Media Handler.”

The following sections provide examples of creating various tween containers and components. For detailed descriptions of the QT atoms used to define tween operations, see “Tween QT Atom Container” (page 960).

Utility Routines

The examples in the next sections use several utility routines to modularize their code. These routines are the following:

- `AddTweenAtom` adds an atom of type `kTweenEntry` plus its standard child atoms (other than the `kTweenDataAtom`) to a container.
- `AddDataAtom` adds a data atom with an ID of `dataAtomID` as a child atom of `tweenAtom`.
- `AddSequenceTweenAtom` adds a `kTweenEntry` atom for a sequenced tween.
- `AddSequenceElement` adds a leaf atom of type `kTweenSequenceElement` to a container.

AddTweenAtom

Listing 25-1 shows `AddTweenAtom`, a routine that adds an atom of type `kTweenEntry` plus its standard child atoms (other than the `kTweenDataAtom`) to a container. Only the `tweenAtomID` and `tweenerType` parameters are required; you may pass 0 for the other parameters to avoid using them.

The `minOutput` and `maxOutput` atom parameters are necessary only if the tweener is being used as an interpolator. Passing a `tweenAtomID` value of 0 means that the routine can assign any unique ID.

If you use `AddTweenAtom` to add a nonsequenced tween entry to a container, the `kTweenEntry` atom it creates is the atom you pass to `QTNewTween` and the sequenceAtom you pass in may be `kParentAtomIsContainer`. In this case the `tweenAtomID` value should be 1.

If you use `AddTweenAtom` to add a sequenced tween entry to a container, the `newSequenceAtom` returned by `AddSequenceTweenAtom` is its `sequenceAtom` parameter and will also be the atom you pass to `QTNewTween`. Note that in most cases a sequenced tween contains only one tween entry but may contain multiple data atoms.

All tween atoms within same `sequenceAtom` must have same tween type. It is unlikely that you would want to change the duration or offset values within the same `sequenceAtom`.

Listing 25-1 Utility routine `AddTweenAtom`

```
OSErr AddTweenAtom( QTAtomContainer container, QTAtom sequenceAtom,
    QTAtomID tweenAtomID, OSType tweenerType, TimeValue offset,
    TimeValue duration, Fixed minOutput, Fixed maxOutput, StringPtr name,
    QTAtom *newTweenAtom )
{
    OSErr       err = noErr;
    QTAtom      tweenAtom = 0;

    if ( ! container )      { err = paramErr; goto bail; }

    err = QTInsertChild( container, sequenceAtom, kTweenEntry,
                            tweenAtomID, 0, 0, nil, &tweenAtom );
    if ( err ) goto bail;

    err = QTInsertChild( container, tweenAtom, kTweenType, 1, 1,
                            sizeof(tweenerType), &tweenerType, nil );
    if ( err ) goto bail;

    if ( offset ) {
        err = QTInsertChild( container, tweenAtom, kTweenStartOffset, 1,
                                1, sizeof(offset), &offset, nil );
        if ( err ) goto bail;
    }

    if ( duration ) {
        err = QTInsertChild( container, tweenAtom, kTweenDuration, 1, 1,
                                sizeof(duration), &duration, nil );
        if ( err ) goto bail;
    }

    // default minOutput is zero, so this is OK
    if ( minOutput ) {
        err = QTInsertChild( container, tweenAtom, kTweenOutputMin, 1, 1,
                                sizeof(minOutput), &minOutput, nil );
        if ( err ) goto bail;
    }

    if ( maxOutput ) {
        err = QTInsertChild( container, tweenAtom, kTweenOutputMax, 1, 1,
                                sizeof(maxOutput), &maxOutput, nil );
        if ( err ) goto bail;
    }

    if ( name ) {
        err = QTInsertChild( container, tweenAtom, kNameAtom, 1, 1,
                                name[0] + 1, name, nil );
        if ( err ) goto bail;
    }

bail:
    if ( newTweenAtom )
        *newTweenAtom = tweenAtom;
    return err;
}
```

AddDataAtom

Listing 25-2 shows `AddDataAtom`, a routine that adds a data atom with an ID of `dataAtomID` as a child atom of `tweenAtom`. If `dataSize` is nonzero, then leaf data is copied from `dataPtr` to `dataAtom`. Otherwise (if `dataContainer` in not `nil`), child atoms are copied from `dataContainer`. If you wish to add the actual data by using another routine, you may pass 0 in `dataSize` and `nil` in both `dataPtr` and `dataContainer`.

You can associate a tweener to be used as an interpolator for each `dataAtom` value. The `interpolationTweenID` parameter specifies the ID of a `kTweenEntry` atom that is a child of the `newSequenceAtom` returned by `AddSequenceTweenAtom`. If you specify an interpolation tweener, then the `atTime` parameter of the `DoTween` routine is first fed as an input to the interpolation tweener. The `tweenResult` of the interpolation tweener becomes the `atTime` parameter of the succeeding tweener. Note that the `kTweenData` atom and the `kTweenInterpolationID` atom have the same ID; this is how QuickTime groups them together.

For best performance, the output range of an interpolation tweener should be from 0 to the duration of the regular tweener. However, you may specify the minimum and maximum values that the interpolation tweener returns; this lets the tweener be shared. The minimum and maximum values are used to scale `tweenResult`, and are added as child atoms of a `kTweenEntry` in `AddTweenAtom`.

Listing 25-2 Utility routine `AddDataAtom`

```
OSErr AddDataAtom( QTAtomContainer container, QTAtom tweenAtom,
    QTAtomID dataAtomID, long dataSize, Ptr dataPtr,
    QTAtomContainer dataContainer, QTAtomID interpolationTweenID,
    QTAtom *newDataAtom )
{
    OSErr       err = noErr;
    QTAtom      dataAtom = 0;

    if ( (! container) || (dataAtomID == 0) || (dataSize &&
        (dataContainer || !dataPtr)) )  { err = paramErr; goto bail; }

    err = QTInsertChild( container, tweenAtom, kTweenData, dataAtomID, 0,
                            dataSize, dataPtr, &dataAtom );
    if ( err ) goto bail;

    if ( dataSize ) {
        err = QTSetAtomData( container, dataAtom, dataSize, dataPtr );
        if ( err ) goto bail;
    }
    else if ( dataContainer ) {
        err = QTInsertChildren( container, dataAtom, dataContainer );
        if ( err ) goto bail;
    }

    if ( interpolationTweenID ) {
        err = QTInsertChild( container, tweenAtom, kTweenInterpolationID,
                            dataAtomID, 0, sizeof(interpolationTweenID),
                            &interpolationTweenID, nil );
        if ( err ) goto bail;
    }

bail:
    if ( newDataAtom )
        *newDataAtom = dataAtom;
    return err;
}
```

AddSequenceTweenAtom

Listing 25-3 shows `AddSequenceTweenAtom`, a routine that adds a `kTweenEntry` atom for a sequenced tween. The `newSequenceAtom` returned may be passed into the `AddTweenAtom` and `AddSequenceElement` routines as their `sequenceAtom` parameter.

To create a nonsequenced tween, use `AddTweenAtom` instead.

Listing 25-3 Utility routine `AddSequenceTweenAtom`

```
OSErr AddSequenceTweenAtom( QTAtomContainer container, QTAtom parentAtom,
                    QTAtomID sequenceAtomID, QTAtom *newSequenceAtom )
{
    if ( (! container) || (sequenceAtomID == 0) )  { return paramErr; }

    return QTInsertChild( container, parentAtom, kTweenEntry,
                        sequenceAtomID, 0, 0, nil, newSequenceAtom );
}
```

AddSequenceElement

Listing 25-4 shows `AddSequenceElement`, a routine that adds a leaf atom of type `kTweenSequenceElement` to a container. The `sequenceAtom` that you pass in is the `newSequenceAtom` parameter returned by `AddSequenceTweenAtom` (page 915). Each time you call `AddSequenceElement`, a sequence tween element is added to the end of the list of elements. The tween toolbox organizes the list in index order, with IDs ignored.

Each element has a duration that is some percentage of the tween's duration. The element's duration is its `endPercent` value minus the previous element's `endPercent` value. For example, if you wanted three elements to last 0.25, 0.25, and 0.5 of the tween's duration, then the elements' `endPercent` values should be set to 0.25, 0.5, and 1. The elements tell the tween toolbox which `tweenAtom` and `dataAtom` to switch to.

The `tweenAtomID` is the ID of a `kTweenEntry` atom within the `sequenceAtom`. The `dataAtomID` is the ID of a `kTweenData` atom. The `kTweenData` atom is a child atom of the specified `tweenAtom`. Usually you only need to create one `tweenAtom` with multiple data atoms. Some tweener types (such as 3D tweeners) use child atoms of the `tweenEntry` atom for initialization, so in these cases you usually create a `tweenAtom` with one `dataAtom` per sequence entry.

Listing 25-4 Utility routine `AddSequenceElement`

```
OSErr AddSequenceElement( QTAtomContainer container, QTAtom sequenceAtom,
    Fixed endPercent, QTAtomID tweenAtomID, QTAtomID dataAtomID,
    QTAtom *newSequenceElementAtom )
{
    TweenSequenceEntryRecord    entry;

    if ( (! container) || (endPercent > (1L<<16)) || (tweenAtomID == 0)
                            || (dataAtomID == 0) ){ return paramErr; }

    entry.endPercent        = endPercent;
    entry.tweenAtomID       = tweenAtomID;
    entry.dataAtomID        = dataAtomID;

    // adds at end of list by index, with any unique atom id
    return QTInsertChild( container, sequenceAtom, kTweenSequenceElement,
                0, 0, sizeof(entry), &entry, newSequenceElementAtom );
}
```

CreateSampleAtomListTweenData

Listing 25-5 shows the `CreateSampleAtomListTweenData` routine, which creates the atom list data used in Listing 25-7 and Listing 25-19.

Listing 25-5 Utility routine `CreateSampleAtomListTweenData`

```
QTAtomContainer CreateSampleAtomListTweenData( long whichOne )
{
    OSErr               err = noErr;
    QTAtomContainer     atomListContainer;
    QTAtomType          tweenAtomType;
    UInt32              elementDataType;
    UInt16              resourceID;

    err = QTNewAtomContainer( &atomListContainer );
    if ( err ) goto bail;

    // kListElementDataType atom specifies the data type of the elements
    elementDataType = kTweenTypeShort;
    err = QTInsertChild( atomListContainer, kParentAtomIsContainer,
                        kListElementDataType, 1, 1,
                        sizeof(tweenAtomType),
                        &tweenAtomType, nil );

    // kListElementType atom tells which type of atoms to look for
    tweenAtomType = 'pcid';
    err = QTInsertChild( atomListContainer, kParentAtomIsContainer,
                        kListElementType, 1, 1, sizeof(tweenAtomType),
                        &tweenAtomType, nil );

    switch ( whichOne ) {
        case 1:
            resourceID = 1000;
            err = QTInsertChild( atomListContainer,
                        kParentAtomIsContainer, 'pcid', 1, 1,
                        sizeof(resourceID), &resourceID, nil );
            if ( err ) goto bail;

            resourceID = 1001;
            err = QTInsertChild( atomListContainer,
                        kParentAtomIsContainer, 'pcid', 2, 2,
                        sizeof(resourceID), &resourceID, nil );
            if ( err ) goto bail;

            resourceID = 1002;
            err = QTInsertChild( atomListContainer,
                        kParentAtomIsContainer, 'pcid', 3, 3,
                        sizeof(resourceID), &resourceID, nil );
            if ( err ) goto bail;
        break;

        case 2:
            resourceID = 1003;
            err = QTInsertChild( atomListContainer,
                        kParentAtomIsContainer, 'pcid', 1, 1,
                        sizeof(resourceID), &resourceID, nil );
            if ( err ) goto bail;

            resourceID = 1004;
            err = QTInsertChild( atomListContainer,
                        kParentAtomIsContainer, 'pcid', 2, 2,
                        sizeof(resourceID), &resourceID, nil );
            if ( err ) goto bail;

            resourceID = 1005;
            err = QTInsertChild( atomListContainer,
                        kParentAtomIsContainer, 'pcid', 3, 3,
                        sizeof(resourceID), &resourceID, nil );
            if ( err ) goto bail;
        break;
    }

bail:
    if ( err && atomListContainer )
        { QTDisposeAtomContainer( atomListContainer );
          atomListContainer = nil; }

    return atomListContainer;
}
```

Creating a Single Tween Container

To create a single tween container, do the following:

1. Create a QT atom container.
2. Insert a `kTweenEntry` atom into the QT atom container for the tween. A `kTweenEntry` atom contains the atoms that define the tween.
3. Insert a `kTweenType` atom that specifies the tween type into the `kTweenEntry` atom.
4. Insert a `kTweenData` atom that contains the data for the tween into the `kTweenEntry` atom .

Listing 25-6 shows how to create a single `kTweenTypeLong` tween container that a component could use to interpolate two long integers.

Listing 25-6 Creating a single `kTweenTypeLong` tween container

```
QTAtomContainer container = nil;
long tweenDataLong[2];
QTAtomType tweenType;
QTAtom tweenAtom;

tweenDataLong[0] = EndianU32_NtoB(512);
tweenDataLong[1] = EndianU32_NtoB(0);

// create a new atom container
QTNewAtomContainer (&container);
// create the parent tween entry atom
tweenType = kTweenTypeLong;
QTInsertChild (container, kParentAtomIsContainer, kTweenEntry, 1, 0, 0,
                nil, &tweenAtom);
// add two child atoms to the tween entry atom --
// * the type atom, kTweenType
QTInsertChild (container, tweenAtom, kTweenType, 1, 0,
                sizeof(tweenType), &tweenType, nil);
// * the data atom, kTweenData
QTInsertChild (container, tweenAtom, kTweenData, 1, 0,
                sizeof(long) * 2, tweenDataLong, nil);
```

Using a List Tween Component

To use a list tween component (of type `kTweenTypeAtomList`), do the following:

1. Create a QT atom container.
2. Insert a `kTweenEntry` atom into the QT atom container for the tween.
3. Insert a `kTweenType` atom that specifies the tween type into the `kTweenEntry` atom.
4. Insert a `kTweenData` atom into the `kTweenEntry` atom.

   Unlike the previous example, this `kTweenData` atom is not a leaf atom.
5. Insert a `kListElementType` atom that specifies the atom type of the list entries into the `kTweenData` atom

   The list entries must be leaf atoms.
6. Insert leaf atoms of the type specified by the `kListElementType` atom into the `kTweenData` atom.

The duration of the tween operation is divided by the number of leaf atoms of the specified type. For time points within the first time division (from the start of the duration up to an including the time (*total time* / *number of atoms*)), the data for the first leaf atom is returned; for the second time division, the data for the second leaf atom is returned; and so on.

Listing 25-7 shows how to create a list tween.

Listing 25-7 Creating a `kTweenTypeAtomList` tween container

```
QTAtomContainer container = nil;
long tweenDataLong[2];
QTAtomType tweenType;
QTAtom tweenAtom;

tweenDataLong[0] = EndianU32_NtoB(512);
tweenDataLong[1] = EndianU32_NtoB(0);

// create a new atom container to hold the sample
QTNewAtomContainer (&container);
// create the parent tween entry atom
tweenType = EndianU32_NtoB(kTweenTypeLong);
QTInsertChild (container, kParentAtomIsContainer, kTweenEntry, 1, 0, 0,
               nil, &tweenAtom);
// add two child atoms to the tween entry atom --
// * the type atom, kTweenType
QTInsertChild (container, tweenAtom, kTweenType, 1, 0,
                sizeof(tweenType), &tweenType, nil);
// * the data atom, kTweenData
QTInsertChild (container, tweenAtom, kTweenData, 1, 0,
                sizeof(short) * 2, tweenDataLong, nil);

OSErr       err = noErr;
QTTweener tween;
QTAtomContainer container = nil, listContainer = nil;
OSType      tweenerType;
TimeValue   offset, duration, tweenTime;
Handle      result;
QTAtom      tweenAtom;

tweenerType = EndianU32_NtoB(kTweenTypeAtomList);
offset  = 0;
duration = 3;

err = QTNewAtomContainer( &container );
if ( err ) goto bail;

err = AddTweenAtom( container, kParentAtomIsContainer, 1, tweenerType,
                    offset, duration, 0, 0, nil, &tweenAtom );
if ( err ) goto bail;

listContainer = CreateSampleAtomListTweenData( 1 );
if ( listContainer == nil ) { err = memFullErr; goto bail; }

err = AddDataAtom( container, tweenAtom, 1, 0, nil,
                    listContainer, 0, nil );
if ( err ) goto bail;

err = QTNewTween( &tween, container, tweenAtom, duration );
if ( err ) goto bail;

result = NewHandle( 0 );
if ( err = MemError() ) goto bail;
    // exercise the AtomListTweener
    for ( tweenTime = 1; tweenTime <= duration; tweenTime += 1 ) {
        long pictureID;
        err = QTDoTween( tween, tweenTime, result, nil, nil, nil );
        if ( err ) goto bail;

        // the pictureID from the atomDataList corresponding to tweenTime
        pictureID = *(long *)*result;
    }

    err = QTDisposeTween( tween );

bail:
    if ( container ) QTDisposeAtomContainer( container );
    if ( listContainer ) QTDisposeAtomContainer( listContainer );
    if ( result ) DisposeHandle( result );
    return err;
}
```

Using a Polygon Tween Component

A polygon tweener maps a four-sided polygon, such as the boundary of a sprite or track, into another. It can be used to create perspective effects, in which the shape of the destination polygon changes over time. The range of polygons into which the source polygon is mapped is defined by two additional four-sided polygons, which are interpolated to specify a destination polygon for any time point in the tween duration.

To use a polygon tween component (of type `kTweenTypePolygon`), do the following:

1. Create a QT atom container.
2. Insert a `kTweenEntry` atom into the QT atom container for the tween.
3. Insert a `kTweenType` atom that specifies the tween type into the `kTweenEntry` atom.
4. Insert a `kTweenData` atom into the `kTweenEntry` atom.

   The data is an array of 27 fixed-point values (`Fixed[27]`) that specifies the three four-sided polygons. Each polygon is specified by 9 consecutive array elements. The first element is each set of 9 contains the number of points used to specify the polygon; this value is coerced to a long integer, and it must always be 4 after coercion. The following 8 values in each set of nine are four `x`, `y` pairs that specify the corners of the polygon.

The first set of 9 elements specifies the dimensions of a sprite or track to be mapped. For example, if the object is a sprite, the four points are `(0,0)`, `(`*`spriteWidth`*`, 0)`, `(`*`spriteWidth`*`, `*`spriteHeight`*`)`, `(0, `*`spriteHeight`*`)`. The next set of 9 elements specifies the initial polygon into which the sprite or track is mapped. The next set of 9 elements specifies the final polygon into which the sprite or track is mapped.

The output is a `MatrixRecord` data structure that you use to map the sprite or track into a four-sided polygon.

Listing 25-8 shows how to create a polygon tween.

Listing 25-8 Creating a polygon tween container

```
OSErr CreateSamplePolygonTweenContainer( QTAtomContainer container,
    TimeValue duration, QTAtom *newTweenAtom )
{
    OSErr           err = noErr;
    TimeValue       offset;
    Handle          thePolygonData = nil;
    QTAtom          tweenAtom;

    err = QTRemoveChildren( container, kParentAtomIsContainer );
    if ( err ) goto bail;

    offset = 0;
    err = AddTweenAtom( container, kParentAtomIsContainer, 1,
                        kTweenTypePolygon, offset, duration, 0, 0,
                        nil, &tweenAtom );
    if ( err ) goto bail;

    thePolygonData = CreateSamplePolygonData();
    if ( thePolygonData == nil ) { err = memFullErr; goto bail; }

    HLock( thePolygonData );

    err = AddDataAtom( container, tweenAtom, 1,
                        GetHandleSize( thePolygonData ),
                        *thePolygonData, nil, 0, nil );
    if ( err ) goto bail;

bail:
    if ( thePolygonData ) DisposeHandle( thePolygonData );
    if ( newTweenAtom ) *newTweenAtom = tweenAtom;
    return err;
}

Handle CreateSamplePolygonData( void )
{
    OSErr       err = noErr;
    Handle      polygonData;
    Fixed       *poly;

    polygonData = NewHandle( 27 * sizeof(Fixed) );
    if ( polygonData == nil ) { err = memFullErr; goto bail; }

    poly = (Fixed *)*polygonData;

    poly[0] = EndianU32_NtoB(4);                // source dimensions
    poly[1] = EndianU32_NtoB(Long2Fix( 0 ));
    poly[2] = EndianU32_NtoB(Long2Fix( 0 ));
    poly[3] = EndianU32_NtoB(Long2Fix( 100 ));
    poly[4] = EndianU32_NtoB(Long2Fix( 0 ));
    poly[5] = EndianU32_NtoB(Long2Fix( 100 ));
    poly[6] = EndianU32_NtoB(Long2Fix( 100 ));
    poly[7] = EndianU32_NtoB(Long2Fix( 0 ));
    poly[8] = EndianU32_NtoB(Long2Fix( 100 ));

    poly[9] = EndianU32_NtoB(4);                // tween from polygon
    poly[10] = EndianU32_NtoB(Long2Fix( 100 ));
    poly[11] = EndianU32_NtoB(Long2Fix( 100 ));
    poly[12] = EndianU32_NtoB(Long2Fix( 200 ));
    poly[13] = EndianU32_NtoB(Long2Fix( 100 ));
    poly[14] = EndianU32_NtoB(Long2Fix( 200 ));
    poly[15] = EndianU32_NtoB(Long2Fix( 200 ));
    poly[16] = EndianU32_NtoB(Long2Fix( 100 ));
    poly[17] = EndianU32_NtoB(Long2Fix( 200 ));

    poly[18] = EndianU32_NtoB(4);               // tween to polygon
    poly[19] = EndianU32_NtoB(Long2Fix( 140 ));
    poly[20] = EndianU32_NtoB(Long2Fix( 100 ));
    poly[21] = EndianU32_NtoB(Long2Fix( 160 ));
    poly[22] = EndianU32_NtoB(Long2Fix( 100 ));
    poly[23] = EndianU32_NtoB(Long2Fix( 200 ));
    poly[24] = EndianU32_NtoB(Long2Fix( 200 ));
    poly[25] = EndianU32_NtoB(Long2Fix( 100 ));
    poly[26] = EndianU32_NtoB(Long2Fix( 200 ));

bail:
    return polygonData;
}
```

Using Path Tween Components

The following sections describe how to use a variety of path tween components. All path tween operations have as input a QuickTime vector data stream for a path. (For information about QuickTime vectors, see Chapter 24, “QuickTime Vectors.”) Path tweeners interpret their time input in one of these ways:

- As a percentage of path’s length. For these types, either a point on the path or a `MatrixRecord` data structure is returned.
- As a function. One operation returns the `y` value of the point on the path with a given `x` value, and the other returns the `x` value of the point on the path with a given `y` value.

If the `kTweenReturnDelta` flag (in an optional `kTweenFlags` atom in the `kTweenEntry` atom) is set, a path tween returns the change in value from the last time it was invoked. If the flag is not set, or if the component has not previously been invoked, the component returns the normal result for the tween.

CreateSampleVectorData Utility

Listing 25-9 shows the `CreateSampleVectorData` routine, which creates the vector data used in Listing 25-10 and Listing 25-20.

Listing 25-9 Utility routine `CreateSampleVectorData`

```
Handle CreateSampleVectorData( long whichOne )
{
    OSErr               err;
```

```
Handle              pathData = nil, vectorData = nil;
ComponentInstance   ci = nil;
gxPoint             aPoint;

err = OpenADefaultComponent( decompressorComponentType,
                                kVectorCodecType, &ci );
if ( err ) goto bail;

err = CurveNewPath( ci, &pathData );
if ( err ) goto bail;

if ( pathData == nil )
    { err = memFullErr; goto bail; }

switch ( whichOne ) {
    case 1:
        aPoint.x = Long2Fix( 0 );
        aPoint.y = Long2Fix( 100 );
        err = CurveInsertPointIntoPath( ci, &aPoint, pathData,
                                        0, 0, false );
        if ( err ) goto bail;

        aPoint.x = Long2Fix( 100 );
        aPoint.y = Long2Fix( 0 );
        err = CurveInsertPointIntoPath( ci, &aPoint, pathData,
                                        0, 1, false );
        if ( err ) goto bail;

        aPoint.x = Long2Fix( 200 );
        aPoint.y = Long2Fix( 100 );
        err = CurveInsertPointIntoPath( ci, &aPoint, pathData,
                                        0, 2, false );
        if ( err ) goto bail;

        aPoint.x = Long2Fix( 100 );
        aPoint.y = Long2Fix( 200 );
        err = CurveInsertPointIntoPath( ci, &aPoint, pathData,
                                        0, 3, false );
        if ( err ) goto bail;
    break;
```

```
        case 2:
            aPoint.x = 0;
            aPoint.y = 100;
            err = CurveInsertPointIntoPath( ci, &aPoint, pathData,
                                            0, 0, false );
            if ( err ) goto bail;

            aPoint.x = 100;
            aPoint.y = 0;
            err = CurveInsertPointIntoPath( ci, &aPoint, pathData,
                                            0, 1, false );
            if ( err ) goto bail;

            aPoint.x = 200;
            aPoint.y = 100;
            err = CurveInsertPointIntoPath( ci, &aPoint, pathData,
                                            0, 2, false );
            if ( err ) goto bail;

            aPoint.x = 100;
            aPoint.y = 200;
            err = CurveInsertPointIntoPath( ci, &aPoint, pathData,
                                            0, 3, false );
            if ( err ) goto bail;
        break;

        case 3:
            aPoint.x = 0;
            aPoint.y = 0;
            err = CurveInsertPointIntoPath( ci, &aPoint, pathData,
                                            0, 0, true );
            if ( err ) goto bail;

            aPoint.x = 200;
            aPoint.y = 50;
            err = CurveInsertPointIntoPath( ci, &aPoint, pathData,
                                            0, 1, true );
            if ( err ) goto bail;

            aPoint.x = 400;
            aPoint.y = 400;
```

```
                err = CurveInsertPointIntoPath( ci, &aPoint, pathData,
                                                0, 2, true );
                if ( err ) goto bail;
            break;

            case 4:
                aPoint.x = Long2Fix( 0 );
                aPoint.y = Long2Fix( 0 );
                err = CurveInsertPointIntoPath( ci, &aPoint, pathData,
                                                0, 0, true );
                if ( err ) goto bail;

                aPoint.x = Long2Fix( 200 );
                aPoint.y = Long2Fix( 50 );
                err = CurveInsertPointIntoPath( ci, &aPoint, pathData,
                                                0, 1, true );
                if ( err ) goto bail;

                aPoint.x = Long2Fix( 400 );
                aPoint.y = Long2Fix( 400 );
                err = CurveInsertPointIntoPath( ci, &aPoint, pathData,
                                                0, 2, true );
                if ( err ) goto bail;
            break;

        }

        err = CurveCreateVectorStream( ci, &vectorData );
        if ( err ) goto bail;

        err = CurveAddPathAtomToVectorStream( ci, pathData, vectorData );
        if ( err ) goto bail;

        err = CurveAddZeroAtomToVectorStream( ci, vectorData );
        if ( err ) goto bail;

    bail:
        if ( pathData ) DisposeHandle( pathData );
        if ( ci ) CloseComponent( ci );
        if ( err != noErr ) {
            if ( vectorData ) {
```

```
            DisposeHandle( vectorData );
            vectorData = nil;
        }
    }
    return vectorData;
}
```

CreateSamplePathTweenContainer Utility

Listing 25-10 shows the `CreateSamplePathTweenContainer` routine, which creates the path tween containers used in Listing 25-11 through Listing 25-19.

Listing 25-10 Utility routine `CreateSamplePathTweenContainer`

```
OSErr CreateSamplePathTweenContainer( QTAtomContainer container,
    OSType tweenerType, long whichSamplePath, Boolean returnDelta,
    TimeValue duration, Fixed initialRotation, QTAtom *newTweenAtom )
{
    OSErr           err = noErr;
    TimeValue       offset;
    Handle          thePathData = nil;
    QTAtom          tweenAtom;

    err = QTRemoveChildren( container, kParentAtomIsContainer );
    if ( err ) goto bail;

    offset = 0;
    err = AddTweenAtom( container, kParentAtomIsContainer, 1,
                        tweenerType, offset, duration, 0, 0, nil,
                        &tweenAtom );
    if ( err ) goto bail;

    thePathData = CreateSampleVectorData( whichSamplePath );
    if ( thePathData == nil ) { err = memFullErr; goto bail; }

    HLock( thePathData );

    err = AddDataAtom( container, tweenAtom, 1,
                        GetHandleSize( thePathData ), *thePathData,
                        nil, 0, nil );
    if ( err ) goto bail;

    if ( returnDelta ) {
        err = AddPathTweenFlags( container, tweenAtom,
                                           kTweenReturnDelta );
    }

    if ( initialRotation )
        QTInsertChild( container, tweenAtom, kInitialRotationAtom,
                1, 1, sizeof(initialRotation), &initialRotation, nil );

bail:
    if ( thePathData ) DisposeHandle( thePathData );
    if ( newTweenAtom ) *newTweenAtom = tweenAtom;
    return err;
```

Using a `kTweenTypePathToMatrixTranslation` Tween Component

To use a `kTweenTypePathToMatrixTranslation` tween component, do the following:

1. Create a QT atom container.
2. Insert a `kTweenEntry` atom into the QT atom container for the tween.
3. Insert a `kTweenType` atom that specifies the tween type into the `kTweenEntry` atom.
4. Insert a `kTweenData` atom into the `kTweenEntry` atom.
5. Perform the tweening operation, using `QTDoTween`.

Listing 25-11 shows how to create a `kTweenTypePathToMatrixTranslation` tween.

Listing 25-11 Creating a `kTweenTypePathToMatrixTranslation` tween container

```
OSErr               err = noErr;
TimeValue           tweenTime, duration;
Handle              result = nil;
QTAtomContainer     container = nil;
QTTweener           tween = nil;
QTAtom              tweenAtom;
```

```
duration = 8;

result = NewHandle( 0 );
if ( err = MemError() ) goto bail;

err = QTNewAtomContainer( &container );
if ( err ) goto bail;

err = CreateSamplePathTweenContainer( container,
                                    kTweenTypePathToMatrixTranslation, 1,
                                    false, duration, 0, &tweenAtom );
if ( err ) goto bail;

err = QTNewTween( &tween, container, tweenAtom, duration );
if ( err ) goto bail;

for ( tweenTime = 0; tweenTime <= duration; tweenTime++ ) {
    MatrixRecord absoluteMatrix;

    err = QTDoTween( tween, tweenTime, result, nil, nil, nil );
    if ( err ) goto bail;

    absoluteMatrix = *(MatrixRecord *)*result;
}

err = QTDisposeTween( tween );

bail:
    if ( container ) QTDisposeAtomContainer( container );
    if ( result ) DisposeHandle( result );
```

Listing 25-12 shows how to create a `kTweenTypePathToMatrixTranslation` tween in which the the `kTweenReturnDelta` flag is set.

Listing 25-12 Creating a `kTweenTypePathToMatrixTranslation` tween

```
err = CreateSamplePathTweenContainer( container,
                                    kTweenTypePathToMatrixTranslation, 1,
                                    true, duration, 0, &tweenAtom );
if ( err ) goto bail;
```

```
err = QTNewTween( &tween, container, tweenAtom, duration );
if ( err ) goto bail;

for ( tweenTime = 0; tweenTime <= duration; tweenTime++ ) {
    MatrixRecord deltaMatrix;

    err = QTDoTween( tween, tweenTime, result, nil, nil, nil );
    if ( err ) goto bail;

    deltaMatrix = *(MatrixRecord *)*result;
}

err = QTDisposeTween( tween );

bail:
    if ( container ) QTDisposeAtomContainer( container );
    if ( result ) DisposeHandle( result );
```

Using a `kTweenTypePathToFixedPoint` Tween Component

To use a `kTweenTypePathToFixedPoint` tween component, do the following:

1. Create a QT atom container.
2. Insert a `kTweenEntry` atom into the QT atom container for the tween.
3. Insert a `kTweenType` atom that specifies the tween type into the `kTweenEntry` atom.
4. Insert a `kTweenData` atom into the `kTweenEntry` atom.
5. Perform the tweening operation, using `QTDoTween`.

Listing 25-13 shows how to create a `kTweenTypePathToFixedPoint` tween.

Listing 25-13 Creating a `kTweenTypePathToFixedPoint` tween container

```
err = CreateSamplePathTweenContainer( container,
                                     kTweenTypePathToFixedPoint, 2, false,
                                     duration, 0, &tweenAtom );
if ( err ) goto bail;
```

```
err = QTNewTween( &tween, container, tweenAtom, duration );
if ( err ) goto bail;

for ( tweenTime = 0; tweenTime <= duration; tweenTime++ ) {
    gxPoint absolutePoint;

    err = QTDoTween( tween, tweenTime, result, nil, nil, nil );
    if ( err ) goto bail;

    absolutePoint = *(gxPoint *)*result;
}

err = QTDisposeTween( tween );

bail:
    if ( container ) QTDisposeAtomContainer( container );
    if ( result ) DisposeHandle( result );
```

Listing 25-14 shows how to create a `kTweenTypePathToFixedPoint` tween in which the `kTweenReturnDelta` flag is set.

Listing 25-14 Creating a `kTweenTypePathToFixedPoint` tween container

```
err = CreateSamplePathTweenContainer( container,
                                    kTweenTypePathToFixedPoint, 2, true,
                                    duration, 0, &tweenAtom );
if ( err ) goto bail;

err = QTNewTween( &tween, container, tweenAtom, duration );
if ( err ) goto bail;

for ( tweenTime = 0; tweenTime <= duration; tweenTime++ ) {
    gxPoint deltaPoint;

    err = QTDoTween( tween, tweenTime, result, nil, nil, nil );
    if ( err ) goto bail;

    deltaPoint = *(gxPoint *)*result;
}
```

```
err = QTDisposeTween( tween );

bail:
    if ( container ) QTDisposeAtomContainer( container );
    if ( result ) DisposeHandle( result );
```

Using a `kTweenTypePathToMatrixRotation` Tween Component

To use a `kTweenTypePathToMatrixRotation` tween component, do the following:

1. Create a QT atom container.
2. Insert a `kTweenEntry` atom into the QT atom container for the tween.
3. Insert a `kTweenType` atom that specifies the tween type into the `kTweenEntry` atom.
4. Insert a `kTweenData` atom into the `kTweenEntry` atom.
5. Perform the tweening operation, using `QTDoTween`.

Listing 25-15 shows how to create a `kTweenTypePathToMatrixRotation` tween.

Listing 25-15 Creating a `kTweenTypePathToMatrixRotation` tween container

```
// kTweenTypePathToMatrixRotation
err = CreateSamplePathTweenContainer( container,
                                kTweenTypePathToMatrixRotation, 1, false,
                                duration, X2Fix(0.5), &tweenAtom );
if ( err ) goto bail;

err = QTNewTween( &tween, container, tweenAtom, duration );
if ( err ) goto bail;

for ( tweenTime = 0; tweenTime <= duration; tweenTime++ ) {
    MatrixRecord absoluteMatrix;

    err = QTDoTween( tween, tweenTime, result, nil, nil, nil );
    if ( err ) goto bail;

    absoluteMatrix = *(MatrixRecord *)*result;
}
```

```
err = QTDisposeTween( tween );

bail:
    if ( container ) QTDisposeAtomContainer( container );
    if ( result ) DisposeHandle( result );
```

Using a `kTweenTypePathToMatrixTranslationAndRotation` Tween Component

To use a `kTweenTypePathToMatrixTranslationAndRotation` tween component, do the following:

1. Create a QT atom container.
2. Insert a `kTweenEntry` atom into the QT atom container for the tween.
3. Insert a `kTweenType` atom that specifies the tween type into the `kTweenEntry` atom.
4. Insert a `kTweenData` atom into the `kTweenEntry` atom.
5. Perform the tweening operation, using `QTDoTween`.

Listing 25-16 shows how to create a `kTweenTypePathToMatrixTranslationAndRotation` tween.

Listing 25-16 Creating a `kTweenTypePathToMatrixTranslationAndRotation` tween container

```
err = CreateSamplePathTweenContainer( container,
                            kTweenTypePathToMatrixTranslationAndRotation,
                            1, false, duration, X2Fix(0.5), &tweenAtom );
if ( err ) goto bail;

err = QTNewTween( &tween, container, tweenAtom, duration );
if ( err ) goto bail;

for ( tweenTime = 0; tweenTime <= duration; tweenTime++ ) {
    MatrixRecord absoluteMatrix;

    err = QTDoTween( tween, tweenTime, result, nil, nil, nil );
    if ( err ) goto bail;
    absoluteMatrix = *(MatrixRecord *)*result;
}

err = QTDisposeTween( tween );

bail:
    if ( container ) QTDisposeAtomContainer( container );
    if ( result ) DisposeHandle( result );
```

Using a `kTweenTypePathXtoY` Tween Component

To use `kTweenTypePathXtoY` tween components, either absolute or delta, do the following:

1. Create a QT atom container.
2. Insert a `kTweenEntry` atom into the QT atom container for the tween.
3. Insert a `kTweenType` atom that specifies the tween type into the `kTweenEntry` atom.
4. Insert a `kTweenData` atom into the `kTweenEntry` atom.
5. Perform the tweening operation, using `QTDoTween`.

Listing 25-17 shows how to create both kinds of `kTweenTypePathXtoY` tweens.

Listing 25-17 Creating `kTweenTypePathXtoY` tweens container

```
// kTweenTypePathXtoY - normal
err = CreateSamplePathTweenContainer( container, kTweenTypePathXtoY, 3,
                                        false, duration, 0, &tweenAtom );
if ( err ) goto bail;

err = QTNewTween( &tween, container, tweenAtom, duration );
if ( err ) goto bail;

for ( tweenTime = 0; tweenTime <= duration; tweenTime++ ) {
    Fixed absoluteYvalue;

    err = QTDoTween( tween, tweenTime, result, nil, nil, nil );
    if ( err ) goto bail;
    absoluteYvalue = *(Fixed *)*result;
}

err = QTDisposeTween( tween );

// kTweenTypePathXtoY - delta
err = CreateSamplePathTweenContainer( container, kTweenTypePathXtoY, 3,
                                        true, duration, 0, &tweenAtom );
if ( err ) goto bail;

err = QTNewTween( &tween, container, tweenAtom, duration );
if ( err ) goto bail;

for ( tweenTime = 0; tweenTime <= duration; tweenTime++ ) {
    Fixed deltaYalue;

    err = QTDoTween( tween, tweenTime, result, nil, nil, nil );
    if ( err ) goto bail;

    deltaYalue = *(Fixed *)*result;
}

err = QTDisposeTween( tween );

bail:
    if ( container ) QTDisposeAtomContainer( container );
    if ( result ) DisposeHandle( result );
```

Using a `kTweenTypePathYtoX` Tween Component

To use `kTweenTypePathYtoX` tween components, either absolute or delta, do the following:

1. Create a QT atom container.
2. Insert a `kTweenEntry` atom into the QT atom container for the tween.
3. Insert a `kTweenType` atom that specifies the tween type into the `kTweenEntry` atom.
4. Insert a `kTweenData` atom into the `kTweenEntry` atom.
5. Perform the tweening operation, using `QTDoTween`.

Listing 25-18 shows how to create both kinds of `kTweenTypePathYtoX` tweens.

Listing 25-18 Creating `kTweenTypePathYtoX` tweens container

```
// kTweenTypePathYtoX - normal
err = CreateSamplePathTweenContainer( container, kTweenTypePathYtoX, 4,
                                        false, duration, 0, &tweenAtom );
if ( err ) goto bail;

err = QTNewTween( &tween, container, tweenAtom, duration );
if ( err ) goto bail;

for ( tweenTime = 0; tweenTime <= duration; tweenTime++ ) {
    Fixed absoluteXvalue;

    err = QTDoTween( tween, tweenTime, result, nil, nil, nil );
    if ( err ) goto bail;

    absoluteXvalue = *(Fixed *)*result;
}

err = QTDisposeTween( tween );

// kTweenTypePathYtoX - delta
err = CreateSamplePathTweenContainer( container, kTweenTypePathYtoX, 4,
                                        true, duration, 0, &tweenAtom );
if ( err ) goto bail;

err = QTNewTween( &tween, container, tweenAtom, duration );
if ( err ) goto bail;

for ( tweenTime = 0; tweenTime <= duration; tweenTime++ ) {
    Fixed deltaXvalue;

    err = QTDoTween( tween, tweenTime, result, nil, nil, nil );
    if ( err ) goto bail;

    deltaXvalue = *(Fixed *)*result;
}
```

```
err = QTDisposeTween( tween );

bail:
    if ( container ) QTDisposeAtomContainer( container );
    if ( result ) DisposeHandle( result );
```

Specifying an Offset for a Tween Operation

You can start a tween operation after a tween media sample begins by including an optional `kTweenStartOffset` atom in the `kTweenEntry` atom for the tween. This atom specifies a time interval, beginning at the start of the tween media sample, after which the tween operation begins. If this atom is not included, the tween operation begins at the start of the tween media sample.

Specifying a Duration for a Tween

You can specify the duration of a tween operation by including an optional `kTweenDuration` atom in the `kTweenEntry` atom for the tween. When a QuickTime movie includes a tween track, the time units for the duration are those of the tween track's time scale. If a tween component is used outside of a movie, the application using the tween data determines how the duration value and values returned by the component are interpreted.

Creating a Tween Sequence

"Single Tweens and Tween Sequences" beginning on page 910 discussed tween sequences, in which different tween operations of the same type may be applied sequentially. The type `kTweenSequenceElement` specifies an entry in a tween sequence. Its parent is the tween QT atom container (which you specify with the constant `kParentAtomIsContainer`).

The ID of a `kTweenSequenceElement` atom must be unique among the `kTweenSequenceElement` atoms in the same QT atom container. The index of a `kTweenSequenceElement` atom specifies its order in the sequence; the first entry in the sequence has the index `1`, the second `2`, and so on.

This atom is a leaf atom. The data type of its data is `TweenSequenceEntryRecord`, a data structure that contains the following fields:

- `endPercent`, a value of type `Fixed` that specifies the point in the duration of the tween media sample at which the sequence entry ends. This is expressed as a fraction; for example, if the value is 0.75, the sequence entry ends after three-quarters of the total duration of the tween media sample has elapsed. The sequence entry begins after the end of the previous sequence entry or, for the first entry in the sequence, at the beginning of the tween media sample.

- `tweenAtomID`, a value of type `QTAtomID` that specifies the `kTweenEntry` atom containing the tween for the sequence element. The `kTweenEntry` atom and the `kTweenSequenceElement` atom must both be child atoms of the same tween QT atom container.
- `dataAtomID`, a value of type `QTAtomID` that specifies the `kTweenData` atom containing the data for the tween. This atom must be a child atom of the atom specified by the `tweenAtomID` field.

Listing 25-19 shows how to create a tween sequence.

Listing 25-19 Creating a tween sequence

```
OSErr CreateSampleSequencedTweenContainer( QTAtomContainer container,
    TimeValue duration, QTAtom *newTweenAtom )
{
    OSErr                err = noErr;
    QTAtomContainer      dataContainer = nil;
    OSType               tweenerType;
    QTAtom               sequenceAtom, tweenAtom;
    TimeValue            offset;
    Handle               result;
    QTAtomID             tweenAtomID, dataAtomID;
    Fixed                endPercent;

    err = QTRemoveChildren( container, kParentAtomIsContainer );
    if ( err ) goto bail;

    tweenerType = kTweenTypeAtomList;
    offset  = 0;

    err = AddSequenceTweenAtom( container, kParentAtomIsContainer,
                                1, &sequenceAtom );
    if ( err ) goto bail;

    offset = 0;

    err = AddTweenAtom( container, sequenceAtom, 1, tweenerType, offset,
                        duration, 0, 0, nil, &tweenAtom );
    if ( err ) goto bail;

    // add first data atom (id 1) to tween atom
    dataAtomID = 1;
    dataContainer = CreateSampleAtomListTweenData( dataAtomID );
    if ( ! dataContainer ) { err = memFullErr; goto bail; }

    err = AddDataAtom( container, tweenAtom, dataAtomID, 0, nil,
                        dataContainer, 0, nil );
    if ( err ) goto bail;

    QTDisposeAtomContainer( dataContainer );

    // add second data atom (id 2) to tween atom
    dataAtomID = 2;
    dataContainer = CreateSampleAtomListTweenData( dataAtomID );
    if ( ! dataContainer ) { err = memFullErr; goto bail; }

    err = AddDataAtom( container, tweenAtom, dataAtomID, 0, nil,
                        dataContainer, 0, nil );
    if ( err ) goto bail;

    QTDisposeAtomContainer( dataContainer );

    // now create a sequence with four elements; the first three are data
    // atom 1, the last is data atom 2
    endPercent = FixDiv( Long2Fix(25), Long2Fix(100) );
    tweenAtomID = 1;
    dataAtomID = 1;
    err = AddSequenceElement( container, sequenceAtom, endPercent,
                                tweenAtomID, dataAtomID, nil );
    if ( err ) goto bail;

    endPercent = FixDiv( Long2Fix(50), Long2Fix(100) );
    tweenAtomID = 1;
    dataAtomID = 1;
    err = AddSequenceElement( container, sequenceAtom, endPercent,
                            tweenAtomID, dataAtomID, nil );
    if ( err ) goto bail;

    endPercent = FixDiv( Long2Fix(75), Long2Fix(100) );
    tweenAtomID = 1;
    dataAtomID = 1;
    err = AddSequenceElement( container, sequenceAtom, endPercent,
                            tweenAtomID, dataAtomID, nil );
    if ( err ) goto bail;

    endPercent = FixDiv( Long2Fix(100), Long2Fix(100) );
    tweenAtomID = 1;
    dataAtomID = 2;
    err = AddSequenceElement( container, sequenceAtom, endPercent,
                            tweenAtomID, dataAtomID, nil );
    if ( err ) goto bail;
bail:
    if ( err ) {
        if ( container )
            QTRemoveChildren( container, kParentAtomIsContainer );
        *newTweenAtom = nil;
    }
    else
        *newTweenAtom = sequenceAtom;
}
```

Creating an Interpolation Tween

"Interpolation Tweens" (page 910) discusses tween operations that modify other tween operations by feeding them artificial time values in place of real time. Listing 25-20 shows how to create an interpolation tween.

Listing 25-20 Creating an interpolation tween container

```
OSErr CreateSampleInterpolatedTweenContainer( QTAtomContainer container,
    TimeValue duration, QTAtom *newTweenAtom )
{
    OSErr          err = noErr;
    Handle         pathData = nil;

    err = QTRemoveChildren( container, kParentAtomIsContainer );
    if ( err )goto bail;

    err = CreateSampleLongTweenContainer( container, 0, duration,
                                        duration, newTweenAtom );
    if ( err ) goto bail;

    pathData =  CreateSampleVectorData( 3 );
    if ( ! pathData ) { err = memFullErr; goto bail; }

    err = AddXtoYInterpolatorTweenerForDataSet( container, *newTweenAtom,
                                        *newTweenAtom, 1, pathData );
    if ( err ) goto bail;
bail:
    if ( pathData )DisposeHandle( pathData );
    return err;
}

OSErr AddXtoYInterpolatorTweenerForDataSet( QTAtomContainer container,
    QTAtom sequenceTweenAtom, QTAtom tweenAtom, QTAtomID dataSetID,
    Handle vectorCodecData )
{
    OSErr              err = noErr;
    QTAtomID           interpolationTweenID;
    QTAtom             dataSetAtom, interpolatorTweenAtom, durationAtom,
                       interpolatorIDAtom;
    TimeValue          duration;
    ComponentInstance  ci = nil;
    UInt8              saveState;
    gxPaths            *thePathData;
    long               dataSize, numPoints;
    gxPoint            firstPoint, lastPoint;
    Boolean            ptIsOnPath;
    Fixed              minOutput, maxOutput;

    if ( (! container) || (! dataSetID) || (! vectorCodecData) )
                                    { err = paramErr; goto bail; }
```

```
saveState = HGetState( vectorCodecData );

dataSetAtom = QTFindChildByID( container, tweenAtom, kTweenData,
                                dataSetID, nil );
if ( ! dataSetAtom ) { err = cannotFindAtomErr; goto bail; }

// determine duration of tweenEntry so we can use the same duration
// for the interpolator tween
durationAtom = QTFindChildByIndex( container, tweenAtom,
                                    kTweenDuration, 1, nil );
if ( ! durationAtom ) { err = cannotFindAtomErr; goto bail; }

err = QTCopyAtomDataToPtr( container, durationAtom, false,
                                sizeof(duration), &duration, nil );
if ( err ) goto bail;

// determine the minOutput and maxOutput values based for the given
// vector codec data
err = OpenADefaultComponent( decompressorComponentType,
                                kVectorCodecType, &ci );
if ( err ) goto bail;

HLock( vectorCodecData );

err = CurveGetAtomDataFromVectorStream ( ci, vectorCodecData,
                    kCurvePathAtom, &dataSize, (Ptr *)&thePathData );
if ( err ) goto bail;

err = CurveCountPointsInPath( ci, thePathData, 0,
                                (unsigned long *)&numPoints );
if ( err ) goto bail;

err = CurveGetPathPoint( ci, thePathData, 0, 0, &firstPoint,
                            &ptIsOnPath );
if ( err ) goto bail;

err = CurveGetPathPoint( ci, thePathData, 0, numPoints - 1,
                            &lastPoint, &ptIsOnPath );
if ( err ) goto bail;
```

```
    minOutput = firstPoint.x;
    maxOutput = lastPoint.x;

    // add interolator tween atom with any unique id
    err = AddTweenAtom( container, sequenceTweenAtom, 0,
                        kTweenTypePathXtoY, 0, duration, minOutput,
                        maxOutput, nil, &interpolatorTweenAtom );
    if ( err ) goto bail;

    // so what was that unique id?
    err = QTGetAtomTypeAndID( container, interpolatorTweenAtom, nil,
                                &interpolationTweenID );
    if ( err ) goto bail;

    err = AddDataAtom( container, interpolatorTweenAtom, 1,
                        GetHandleSize( vectorCodecData ),
                        *vectorCodecData, nil, 0, nil );
    if ( err ) goto bail;

    // finally, we need to reference this new interpolator tween
    interpolatorIDAtom  =  QTFindChildByID( container, tweenAtom,
                            kTweenInterpolationID, dataSetID, nil );
    if ( ! interpolatorIDAtom ) {
        err = QTInsertChild( container, tweenAtom, kTweenInterpolationID,
                            dataSetID, 0, 0, nil, &interpolatorIDAtom );
        if ( err ) goto bail;
    }

    err = QTSetAtomData( container, interpolatorIDAtom,
                sizeof(interpolationTweenID), &interpolationTweenID );
    if ( err ) goto bail;

bail:
    if ( vectorCodecData )
        HSetState( vectorCodecData, saveState );
    return err;
}
```

To scale the output of an interpolation tween, you add the optional `kTweenOutputMaxValue` atom and `kTweenOutputMinValue` atom.

Naming Tweens

You can use the kNameAtom atom to store a string value containing a name (or any other information) in a tween container. This atom is not required and is not routinely accessed by QuickTime. It is available for use by your authoring tools or other software.

Creating a Tween Component

This section describes the functions you must implement when creating a tween component. Before reading this section, you should be familiar with how to create components. See "Component Manager" in *Mac OS For QuickTime Programmers* for a complete description of components, how to use them, and how to create them.

The following example illustrates a tween component that interpolates values for short integers. Because QuickTime handles this tween type (kTweenTypeShort) for you, you do not need to implement a component to handle interpolation of short integers yourself.

Initializing the Tween Component

Your tween component must process kTweenerInitializeSelect requests from the Component Manager. Listing 25-21 shows a function, TweenerInitialize, for processing this request. In this example, the function simply returns. In a more complex example, the function might allocate storage to be used when generating a tween media value.

Listing 25-21 Function that initializes a tween component

```
pascal ComponentResult TweenerInitialize (
                                                        TweenerComponent tc,
                                                        QTAtomContainer container,
                                                        QTAtom tweenAtom,
                                                        QTAtom dataAtom)
```

```
{
    return noErr;
}
```

Generating Tween Media Values

Your tween component must process `kTweenerDoTweenSelect` requests from the Component Manager. Listing 25-22 shows a function, `TweenDoTween`, for processing this request. It takes short-integer values and performs the necessary interpolation.

Listing 25-22 Function that generates tween media values

```
pascal ComponentResult TweenDoTween (
                                        TweenerComponent tc,
                                        TweenRecord *tr)
{
    short       *data;
    short       tFrom, tTo, tValue;

    QTGetAtomDataPtr(tr->container, tr->dataAtom, nil, (Ptr *)&data);

    tFrom = data[0];
    tTo = data[1];
    tValue = tFrom + FixMul(tTo - tFrom, tr->percent);

    (tr->dataProc)((struct TweenRecord *)tr, &tValue,
        sizeof(tValue), 1, nil, nil, nil, nil);

    return noErr;
}
```

Resetting a Tween Component

Your tween component must process `kTweenerResetSelect` requests from the Component Manager. Listing 25-23 shows the `TweenReset` function, which resets the component. In this example, because `TweenerInitialize` does not allocate any storage, `TweenerReset` simply returns. In a more complex example, TweenerReset releases any storage allocated by TweenerInitialize and any storage allocated during the tween operation.

Listing 25-23 Function that resets a tween component

```
pascal ComponentResult TweenerReset (TweenerComponent tc)
{
    return noErr;
}
```

Tween Component and Native Tween Type Reference

This section describes the native tween types handled by QuickTime; the tween components included in QuickTime; and the constants, data types, and routines associated with tween components.

Tween Container Syntax

Tween containers conform to the following syntax:

```
[(TweenContainerFormat)] =  [(SingleTweenFormat)] |
[(SequencedTweenFormat)]

[(SingleTweenFormat)] =
[(TweenEntryAtoms)]

<kTweenEntry>, (anyUniqueIDs), (1..numInterpolators)
[(TweenEntryAtoms)]

[(SequencedTweenFormat)] =
kTweenSequenceElement, (anyUniqueIDs), (1..numSequenceElements)
[TweenSequenceEntryRecord] = {endPercent, tweenAtomID, dataAtomID}

kTweenEntry, (anyUniqueIDs), (1..numSequenceElements + numInterpolators)
[(TweenEntryAtoms)]

[(TweenEntryAtoms)] =
kTweenType, 1, 1
[OSType] = the type of tween
```

```
<kTweenStartOffset>, 1, 1
[TimeValue] = starting offset

<kTweenDuration>, 1, 1
[TimeValue] = duration

<kTweenOutputMinValue>, 1, 1
[Fixed] = minimum output value

<kTweenOutputMaxValue>, 1, 1
[Fixed] = maximum output value

<kTweenFlags>, 1, 1
[long] = flags

kTweenData, (anyUniqueIDs), (1..numDataAtoms)
// contents dependent on kTweenType, could be leaf data or nested atoms

<kTweenInterpolationID>, (a kTweenData ID), (1.. numInterpolationIDAtoms)
[QTAtomID] = the id of a kTweenEntry (child of [(TweenContainerFormat)]
describing the tween to be used to interpolate time values.
```

IMPORTANT

`QTAtomContainer`-based data structures are being widely used in QuickTime. For more information about this syntax and these data structures, see Appendix D. This appendix provides you with a standardized format that describes how these data structures may be expressed and how they are currently documented. ▲

Native Tween Types Processed by QuickTime

A number of tween types are processed by QuickTime itself rather than by separate components. These tween types all produce linear interpolations. Their inputs and outputs are described in the following sections.

kTweenTypeFixed

Input data: Two 32-bit fixed-point values.

Output data: A 32-bit fixed-point value.

kTweenTypeFixedPoint

Input data: Two structures of type `FixedPoint` that describe QuickDraw points.

Output data: A structure of type `FixedPoint` that describes a QuickDraw point.

How interpolation is performed: Each of the two coordinate values used to specify a fixed point is interpolated separately from the other.

kTweenTypeGraphicsModeWithRGBColor

Input data: Two `ModifierTrackGraphicsModeRecord` data structures.

Output data: A `ModifierTrackGraphicsModeRecord` data structure.

How interpolation is performed: Only the `RGBColor` fields of the `ModifierTrackGraphicsModeRecord` data structures are interpolated. The graphic mode used is the first graphic mode that is specified in the modifier track.

kTweenTypeLong

Input data: Two signed 32-bit integers.

Output data: A signed 32-bit integer.

kTweenTypeMatrix

Input data: Two QuickTime 3X3 matrices (data structures of type `MatrixRecord`).

Output data: A QuickTime 3X3 matrix (a data structure of type `MatrixRecord`).

How interpolation is performed: Each of the individual matrix elements is interpolated separately from the other.

kTweenTypePoint

Input data: Two QuickDraw points (data structures of type `Point`).

Output data: A QuickDraw point (a data structure of type `Point`).

How interpolation is performed: Each of the two coordinate values used to specify a QuickDraw point (`h` and `v`) is interpolated separately from the other.

kTweenTypeQDRect

Input data: Two QuickDraw rectangles (data structures of type `Rect`).

Output data: A QuickDraw rectangle (a data structure of type `Rect`).

How interpolation is performed: Each of the four coordinate values used to specify a QuickDraw rectangle (`r`, `l`, `b`, `t`) is interpolated separately from the others.

kTweenTypeQDRegion

Input data: Two QuickDraw rectangles (data structures of type `Rect`) and either a `kTweenRegionData` atom that contains a QuickDraw region (a data structure of type `Region`) or a `kTweenPictureData` atom that contains a QuickDraw picture (a data structure of type `Picture`).

Output data: A QuickDraw region (a data structure of type `Region`).

How interpolation is performed: The two QuickDraw rectangles are interpolated as in a `kTweenTypeQDRect` tween: each of the four coordinate values used to specify a QuickDraw rectangle (`r`, `l`, `b`, `t`) is interpolated separately from the others. If the input data includes a `kTweenRegionData` atom, the QuickDraw region contained in the atom is mapped into the resulting rectangle. If the input data includes a `kTweenPictureData` atom, the QuickDraw picture contained in the atom is drawn into the resulting rectangle.

kTweenTypeQTFloatDouble

Input data: Two double-precision (64-bit) IEEE floating-point numbers of type `double`.

Output data: A double-precision (64-bit) IEEE floating-point number of type `double`.

kTweenTypeQTFloatSingle

Input data: Two single-precision floating-point numbers of type `float`.

Output data: A single-precision floating-point number of type `float`.

kTweenTypeRGBColor

Input data: Two RGB colors (data structures of type `RGBColor`).

Output data: An RGB color (a data structure of type `RGBColor`).

How interpolation is performed: Each of the three values used to specify an RGB color (`red`, `green`, `blue`) is interpolated separately from the others.

kTweenTypeShort

Input data: Two signed 16-bit integers.

Output data: A signed 16-bit integer.

Tween Components for QuickDraw 3D

QuickTime includes a number of components that perform tween operations on QuickDraw 3D objects. These components are described in the following sections. Each component processes two input values and returns one output value, all of the same type. The tween algorithm in all cases performs a visually smooth interpolation between the two objects. The tween types and the QuickDraw 3D data structures that they interpolate are the following:

Tween type	QuickDraw 3D data structure
`kTweenType3dCameraData`	`TQ3CameraData`
`kTweenType3dMatrix`	`TQ3Matrix4x4`
`kTweenType3dMatrixNonLinear`	`TQ3Matrix4x4`
`kTweenType3dQuaternion`	`TQ3Quaternion`
`kTweenType3dRotate`	`TQ3RotateTransformData`
`kTweenType3dRotateAboutAxis`	`TQ3RotateAboutAxisTransformData`
`kTweenType3dRotateAboutPoint`	`TQ3RotateAboutPointTransformData`
`kTweenType3dRotateAboutVector`	`TQ3RotateTransformData`
`kTweenType3dScale`	`TQ3Vector3D`
`kTweenType3dSoundLocalizationData`	`SSpLocalizationData`
`kTweenType3dTranslate`	`TQ3Vector3D`

Other Tween Components

QuickTime includes a number of other components for processing tweens. These components are described in the following sections. Each component processes one or more input values contained in the tween media and returns output values. The description of each tween component lists the data types of the component's input and output data and the number of input values that the component processes.

List Tweener Components

A component of type `kTweenTypeAtomList` is the component that processes list tweens. For an introduction to list tweens, see "Using a List Tween Component" (page 919).

Input data: A QT atom list. This is a `kTweenData` atom that contains

- a `kListElementType` atom that specifies the atom type of the elements and the output
- one or more leaf atoms, ordered by their index values, of the type specified by the `kListElementType` atom
- atoms of other types, which are optional and ignored by the component; these optional atoms can be used by an application, such as atoms that specify a name for each element.

Output data: The data for one of the list element atoms. How this atom is determined is described in "How interpolation is performed."

How interpolation is performed: The duration of the tween is divided by the number of elements in the list. At the time point for which a result is to be returned, the component determines the list element for which to return data. If (0 <= *time value* <= (1 * (*tween duration* / *number of list elements*)), the component returns the data for the first list element; if ((1 * (*tween duration* / *number of list elements* < *time value* <= (2 * (*tween duration* / *number of list elements*)), the component returns the data for the second list element , and so on. For example, if the tween duration is 100 and there are 10 elements in the list, the component returns the value of the first element for a time from 0 to 10, the value of the second element is returned for a time greater than 10 and less than or equal to 20, and so on. The total time for the tween is divided equally among the list elements.

The `kTweenTypeAtomList` container description is the following:

```
[(QTAtomListTweenEntryAtoms)] =
    kListElementType, 1, 1
    // a QTAtomType specifying the type of atoms that make up the list

    kListElementDataType, 1, 1
    //a UInt32 that contains one of the allowed kTweenType flags. kTweenTypeShort
    // through kTweenTypeFixedPoint are allowed]

    kTweenData, 1, 1
        kTweenType, 1, 1
        // QTAtomType for elements in list, for example 'pcid'
        'pcid', anyUniqueID, 1
            [data for first element]
        'pcid', anyUniqueID, 2
        // data for second element
        ...
        'pcid', anyUniqueID, n
        // data for nth element

<kTweenStartOffset>, 1, 1
[TimeValue] = starting offset

<kTweenDuration>, 1, 1
[TimeValue] = duration

<kTweenOutputMinValue>, 1, 1
[Fixed] = minimum output value

<kTweenOutputMaxValue>, 1, 1
[Fixed] = maximum output value

<kTweenSequenceElement>,  (anyUniqueIDs), (1..numElementsInSequence)
[TweenSequenceEntryRecord]

<kTweenInterpolationID>, (a kTweenData ID), (1.. numInterpolationIDAtoms)
[QTAtomID] = the id of a kTweenEntry (child of [(TweenContainerFormat)]
describing the tween to be used to interpolate time values.
```

Multimatrix Tweener Component

The multimatrix tweener, of type `kTweenTypeMultiMatrix`, returns a `MatrixRecord` that is a concatenation of several matrix tweeners. This record can be applied to a sprite or track.

An example of using the multimatrix tweener would be to make a sprite follow a path using the path tweener and at the same time apply a distortion effect using the polygon tweener.

Input data: A list of `kTweenEntry` atoms.

Output data: a matrix record

How interpolation is performed: The data for the tweener consists of a list of `kTweenEntry` atoms, each containing `[(QTAtomListEntryAtoms)]` for any type of tweener that returns a matrix. The order of matrix concatenation is important; the matrices are applied in the order determined by index of the `kTweenEntry` child atoms of the multimatrix tweener's data atom.

Path Tweener Components

A path tweener component returns a point along a path depending on the current time value. The point is either returned as a `FixedPoint` value or a `MatrixRecord` with *x* and *y* offsets corresponding to the point. There are six component subtypes:

- Subtype `kTweenTypePathToFixedPoint` returns a `tweenResult` of type `FixedPoint`.
- Subtype `kTweenTypePathToMatrixTranslation` returns a `tweenResult` of type `MatrixRecord` and performs translation tweening.
- Subtype `kTweenTypePathToMatrixRotation` returns a `tweenResult` of type `MatrixRecord` and performs rotation tweening.
- Subtype `kTweenTypePathToMatrixTranslationAndRotation` returns a `tweenResult` of type `MatrixRecord` and performs both translation and rotation tweening.
- Subtype `kTweenTypePathXtoY` returns a `tweenResult` of type `FixedPoint` that specifies the *y*-coordinate value for a given *x*-coordinate value.
- Subtype `kTweenTypePathYtoX` returns a `tweenResult` of type `FixedPoint` that specifies the *x*-coordinate value for a given *y*-coordinate value.

An example of using a path tweener would be to store a path that you want a sprite to follow. The `MatrixRecord` returned could be used to determine the offset of the sprite.

The path tweener's path format is the one used by the QuickTime vector codec. A transcoder exists that converts a QuickDraw GX shape into this format.

This component uses only the first contour of the first path in the vector data when determining the output. It ignores any additional atoms and paths in the vector data.

Note
If the `kTweenReturnDelta` flag (in an optional `kTweenFlags` atom) is set, the component returns the change in value from the last time it was invoked. If the flag is not set, or if the component has not previously been invoked, the component returns the normal result for the tween. ◆

Polygon Tweener Component

A component of type `kTweenTypePolygon` tweens one polygon into another. All input polygons must be convex and not self-intersecting.

Input data: An array of 27 fixed-point values (`Fixed[27]`) that specifies three four-sided polygons. Each polygon is specified by 9 consecutive array elements. The first element in each set of 9 contains the number of points used to specify the polygon; this value is coerced to a long integer, and it must always be 4 after coercion. The following 8 values in each set of nine are four `x`, `y` pairs that specify the corners of the polygon.

The first set of 9 elements specifies the dimensions of a sprite or track to be mapped. For example, if the object is a sprite, the four points are `(0,0)`, `(`*`spriteWidth`*`, 0)`, `(`*`spriteWidth`*`, `*`spriteHeight`*`)`, `(0, `*`spriteHeight`*`)`. The next set of 9 elements specifies the initial polygon into which the sprite or track is mapped. The next set of 9 elements specifies the ending polygon into which the sprite or track is mapped.

Output data: A MatrixRecord that can be used to map the sprite or track into a four-sided polygon. During the duration of the tween, the shape of this polygon is transformed linearly from that of the initial polygon specified in the input data to that of the ending polygon specified in the input data.

Spin Tweener Component

A component of type `kTweenTypeSpin` returns a `MatrixRecord` that can be applied to a sprite or a track. The matrix returned causes a rotation based on a given number of rotations over the duration of the tween. The data for the tweener consists of an array of two `Fixed` numbers. The first `Fixed` number is the `intialRotation` value, specified as a fraction of one rotation. A number between 0 and 1 is expected; for instance, a value of 0.25 represents a rotation of 90 degrees. The second `Fixed` number is the number of rotations that should occur over the durationof the tween. For instance, to spin a sprite four and a half times this number should be 4.5.

Note
The rotation is performed about an object's origin, which is usually 0.0. For a sprite, its origin is defined by its registration point; hence a spinning sprite will spin about its registration point. For more information about sprites, see *Programming with QuickTime Sprites.* ◆

The spin tweener container description is the following:

```
[SpinTweenEntryAtoms)] =
    kTweenType, 1, 1
        [kTweenTypeSpin]

    kTweenData, 1, 1
        Fixed[2]

<kTweenStartOffset>, 1, 1
[TimeValue] = starting offset

<kTweenDuration>, 1, 1
[TimeValue] = duration

<kTweenSequenceElement>,  (anyUniqueIDs), (1..numElementsInSequence)
[TweenSequenceEntryRecord]

<kTweenInterpolationID>, (a kTweenData ID), (1.. numInterpolationIDAtoms)
[QTAtomID] = the id of a kTweenEntry (child of [(TweenContainerFormat)]
describing the tween to be used to interpolate time values.
```

Constants

Tween Component Constant

The `TweenComponentType` constant specifies that the component is a tween component.

```
enum {
    TweenComponentType  = 'twen'
};
```

Tween Type and Tween Component Subtype Constants

The following constants specify tween types. If a tween type is identified by a four-character code, the four-character code is also the component subtype of the tween component that is invoked for the tween. If a tween type is identified by a numeric constant, such as `kTweenTypeShort`, tweens of that type are processed by QuickTime rather than by a tween component.

```
enum {
    kTweenTypeShort = 1,
    kTweenTypeLong = 2,
    kTweenTypeFixed = 3,
    kTweenTypePoint = 4,
    kTweenTypeQDRect = 5,
    kTweenTypeQDRegion = 6,
    kTweenTypeMatrix = 7,
    kTweenTypeRGBColor = 8,
    kTweenTypeGraphicsModeWithRGBColor  = 9,
    kTweenTypeQTFloatSingle = 10,
    kTweenTypeQTFloatDouble = 11,
    kTweenTypeFixedPoint = 12,
    kTweenType3dScale = FOUR_CHAR_CODE('3sca'),
    kTweenType3dTranslate = FOUR_CHAR_CODE('3tra'),
    kTweenType3dRotate = FOUR_CHAR_CODE('3rot'),
    kTweenType3dRotateAboutPoint  = FOUR_CHAR_CODE('3rap'),
    kTweenType3dRotateAboutAxis = FOUR_CHAR_CODE('3rax'),
    kTweenType3dRotateAboutVector  = FOUR_CHAR_CODE('3rvc'),
    kTweenType3dQuaternion = FOUR_CHAR_CODE('3qua'),
    kTweenType3dMatrix = FOUR_CHAR_CODE('3mat'),
    kTweenType3dCameraData = FOUR_CHAR_CODE('3cam'),
    kTweenType3dSoundLocalizationData  = FOUR_CHAR_CODE('3slc'),
    kTweenTypePathToMatrixTranslation  = FOUR_CHAR_CODE('gxmt'),
    kTweenTypePathToMatrixTranslationAndRotation =FOUR_CHAR_CODE('gxmr'),
    kTweenTypePathToFixedPoint = FOUR_CHAR_CODE('gxfp'),
    kTweenTypePathXtoY = FOUR_CHAR_CODE('gxxy'),
    kTweenTypePathYtoX = FOUR_CHAR_CODE('gxyx'),
    kTweenTypeAtomList = FOUR_CHAR_CODE('atom'),
    kTweenTypePolygon = FOUR_CHAR_CODE('poly')
    kTweenTypePathToMatrixRotation  = FOUR_CHAR_CODE('gxpr'),
    kTweenTypeMultiMatrix = FOUR_CHAR_CODE('mulm'),
    kTweenTypeSpin = FOUR_CHAR_CODE('spin'),
    kTweenType3dMatrixNonLinear = FOUR_CHAR_CODE('3nlr'),
    kTweenType3dVRObject = FOUR_CHAR_CODE('3vro')
    };
```

Tween Atom Constants

The following constants are defined for tween-related atoms. These atom types are described in “Tween QT Atom Container” (page 960).

```
enum {
    kTweenEntry                 = FOUR_CHAR_CODE('twen'),
    kTweenData                  = FOUR_CHAR_CODE('data'),
    kTweenType                  = FOUR_CHAR_CODE('twnt'),
    kTweenStartOffset           = FOUR_CHAR_CODE('twst'),
    kTweenDuration              = FOUR_CHAR_CODE('twdu'),
    kTweenFlags                 = FOUR_CHAR_CODE('flag'),
    kTweenOutputMin             = FOUR_CHAR_CODE('omin'),
    kTweenOutputMax             = FOUR_CHAR_CODE('omax'),
    kTweenSequenceElement       = FOUR_CHAR_CODE('seqe'),
    kTween3dInitialCondition    = FOUR_CHAR_CODE('icnd'),
    kTweenInterpolationID       = FOUR_CHAR_CODE('intr'),
    kTweenRegionData            = FOUR_CHAR_CODE('qdrg'),
    kTweenPictureData           = FOUR_CHAR_CODE('PICT'),
    kListElementType            = FOUR_CHAR_CODE('type'),
    kNameAtom                   = FOUR_CHAR_CODE('name'),
    kInitialRotationAtom        = FOUR_CHAR_CODE('inro')
};
```

Tween Flags

The following flags modify the characteristics of a tween.

```
enum {
    kTweenReturnDelta= 1L << 0
};
```

The `kTweenReturnDelta` flag applies only to path tweens (tweens of type `kTweenTypePathToFixedPoint`, `kTweenTypePathToMatrixTranslation`, `kTweenTypePathToMatrixTranslationAndRotation`, `kTweenTypePathXtoY`, or `kTweenTypePathYtoX`). If the flag is set, the tween component returns the change in value from the last time it was invoked. If the flag is not set, or if the tween component has not previously been invoked, the tween component returns the normal result for the tween.

Data Types

The following sections describe the component instance definition, tween record, and the value setting prototype function used by tween components.

Tween QT Atom Container

The characteristics of a tween are specified by the atoms in a tween QT atom container. For information about QT atom containers, see "QuickTime Atoms" (page 47).

A tween QT atom container can contain the following atoms:

General Tween Atoms

- `kTweenEntry`

  Specifies a tween, which can be either a single tween, a tween in a tween sequence, or an interpolation tween.

  Its parent is the tween QT atom container (which you specify with the constant `kParentAtomIsContainer`).

  The index of a `kTweenEntry` atom specifies when it was added to the QT atom container; the first added has the index `1`, the second `2`, and so on. The ID of a `kTweenEntry` atom can be any ID that is unique among the `kTweenEntry` atoms contained in the same QuickTime atom container.

This atom is a parent atom. It must contain the following child atoms:

- A `kTweenType` atom that specifies the tween type.
- One or more `kTweenData` atoms that contain the data for the tween. Each `kTweenData` atom can contain different data to be processed by the tween component, and a tween component can process data from only one `kTweenData` atom a time. For example, an application can use a list tween to animate sprites. The `kTweenEntry` atom for the tween could contain three sets of animation data, one for moving the sprite from left to right, one for moving the sprite from right to left, and one for moving the sprite from top to bottom. In this case, the `kTweenEntry` atom for the tween would contain three `kTweenData` atoms, one for each data set. The application specifies the desired data set by specifying the ID of the `kTweenData` atom to use.

A `kTweenEntry` atom can contain any of the following optional child atoms:

- A `kTweenStartOffset` atom that specifies a time interval, beginning at the start of the tween media sample, after which the tween operation begins. If this atom is not included, the tween operation begins at the start of the tween media sample.
- A `kTweenDuration` atom that specifies the duration of the tween operation. If this atom is not included, the duration of the tween operation is the duration of the media sample that contains it.

If this atom specifies a path tween, it can contain the following optional child atom:

- A `kTweenFlags` atom containing flags that control the tween operation. If this atom is not included, no flags are set.

If a `kTweenEntry` atom specifies an interpolation tween, it must contain the following child atom(s):

- A `kTweenInterpolationID` atom for each `kTweenData` atom to be interpolated. The ID of each `kTweenInterpolationID` atom must match the ID of the `kTweenData` atom to be interpolated. The data for a `kTweenInterpolationID` atom specifies a `kTweenEntry` atom that contains the interpolation tween to use for the `kTweenData` atom.

If this atom specifies an interpolation tween, it can contain either of the following optional child atoms:

- A `kTweenOutputMin` atom that specifies the minimum output value of the interpolation tween. The value of this atom is used only if there is also a `kTweenOutputMax` atom with the same parent. If this atom is not included and there is a `kTweenOutputMax` atom with the same parent, the tween component uses `0` as the minimum value when scaling output values of the interpolation tween.

- □ A `kTweenOutputMax` atom that specifies the maxiumum output value of the interpolation tween. If this atom is not included, the tween component does not scale the output values of the interpolation tween.

- `kTweenStartOffset`

  For a tween in a tween track of a QuickTime movie, specifies a time offset from the start of the tween media sample to the start of the tween. The time units are the units used for the tween track.

  Its parent atom is a `kTweenEntry` atom.

  A `kTweenEntry` atom can contain only one `kTweenStartOffset` atom. The ID of this atom is always `1`. The index of this atom is always `1`.

  This atom is a leaf atom. The data type of its data is `TimeValue`.

  This atom is optional. If it is not included, the tween operation begins at the start of the tween media sample.

- `kTweenDuration`

  Specifies the duration of a tween operation. When a QuickTime movie includes a tween track, the time units for the duration are those of the tween track. If a tween component is used outside of a movie, the application using the tween data determines how the duration value and values returned by the component are interpreted.

  Its parent atom is a `kTweenEntry` atom.

  A `kTweenEntry` atom can contain only one `kTweenDuration` atom. The ID of this atom is always `1`. The index of this atom is always `1`.

  This atom is a leaf atom. The data type of its data is `TimeValue`.

  This atom is optional. If it is not included, the duration of the tween is the duration of the media sample that contains it.

- `kTweenData`

  Contains data for a tween.

  Its parent atom is a `kTweenEntry` atom.

  A `kTweenEntry` atom can contain any number of `kTweenData` atoms. Each `kTweenData` atom can contain different data to be processed by the tween component, and a tween component can process data from only one `kTweenData` atom a time. For example, an application can use a list tween to animate sprites. The `kTweenEntry` atom for the tween could contain three sets of animation data, one for moving the sprite from left to right, one for moving the sprite from right to left, and one for moving the sprite from top to bottom. In this case, the `kTweenEntry` atom for the tween would contain three `kTweenData` atoms, one for each data set. The application would specify the desired data set by specifying the ID of the `kTweenData` atom to use.

The index of a `kTweenData` atom specifies when it was added to the `kTweenEntry` atom; the first added has the index `1`, the second `2`, and so on. The ID of a `kTweenData` atom can be any ID that is unique among the `kTweenData` atoms contained in the same `kTweenEntry` atom.

At least one `kTweenData` atom is required in a `kTweenEntry` atom.

For single tweens, a `kTweenData` atom is a leaf atom. It can contain data of any type.

For polygon tweens, a `kTweenData` atom is a leaf atom. The data type of its data is `Fixed[27]`, which specifies three polygons as described in "Using Path Tween Components" (page 925).

For path tweens, a `kTweenData` atom is a leaf atom. The data type of its data is `Handle`, which contains a QuickTime vector as described in "Using Path Tween Components" (page 925).

In interpolation tweens, a `kTweenData` atom is a leaf atom. It can contain data of any type. An interpolation tween can be any tween other than a list tween that returns a time value, as described in "Interpolation Tweens" (page 910).

In list tweens, a `kTweenData` atom is a parent atom that must contain the following child atoms:

□ A `kListElementType` atom that specifies the atom type of the elements iof the tween.

□ One or more leaf atoms of the type specified by the `kListElementType` atom. The data for each atom is the result of a list tween operation, as described in "Using a List Tween Component" (page 919).

■ `kNameAtom`

Specifies the name of a tween. The name, which is optional, is not used by tween components, but it can be used by applications or other software.

Its parent atom is a `kTweenEntry` atom.

A `kTweenEntry` atom can contain only one `kNameAtom` atom. The ID of this atom is always `1`. The index of this atom is always `1`.

This atom is a leaf atom. Its data type is `StringPtr`.

This atom is optional. If it is not included, the tween does not have name.

- `kTweenType`

  Specifies the tween type (the data type of the data for the tween operation).

  Its parent atom is a `kTweenEntry` atom.

  A `kTweenEntry` atom can contain only one `kTweenType` atom. The ID of this atom is always `1`. The index of this atom is always `1`.

  This atom is a leaf atom. The data type of its data is `OSType`.

  This atom is required.

Path Tween Atoms

- `kTweenFlags`

  Contains flags that control the tween operation. One flag that controls path tweens is defined:

  - The `kTweenReturnDelta` flag applies only to path tweens (tweens of type `kTweenTypePathToFixedPoint`, `kTweenTypePathToMatrixTranslation`, `kTweenTypePathToMatrixTranslationAndRotation`, `kTweenTypePathXtoY`, or `kTweenTypePathYtoX`). If the flag is set, the tween component returns the change in value from the last time it was invoked. If the flag is not set, or if the tween component has not previously been invoked, the tween component returns the normal result for the tween.

  Its parent atom is a `kTweenEntry` atom.

  A `kTweenEntry` atom can contain only one `kTweenFlags` atom. The ID of this atom is always `1`. The index of this atom is always `1`.

  This atom is a leaf atom. The data type of its data is `Long`.

  This atom is optional. If it is not included, no flags are set.

- `kInitialRotationAtom`

  Specifies an initial angle of rotation for a path tween of type `kTweenTypePathToMatrixRotation`, `kTweenTypePathToMatrixTranslation`, or `kTweenTypePathToMatrixTranslationAndRotation`.

  Its parent atom is a `kTweenEntry` atom.

  A `kTweenEntry` atom can contain only one `kInitialRotationAtom` atom. The ID of this atom is always `1`. The index of this atom is always `1`.

  This atom is a leaf atom. Its data type is `Fixed`.

  This atom is optional. If it is not included, no initial rotation of the tween is performed.

List Tween Atoms

- `kListElementType`

  Specifies the atom type of the elements in a list tween.

  Its parent atom is a `kTweenData` atom.

  A `kTweenEntry` atom can contain only one `kListElementType` atom. The ID of this atom is always `1`. The index of this atom is always `1`.

  This atom is a leaf atom. Its data type is `QTAtomType`.

  This atom is required in the `kTweenData` atom for a list tween.

3D Tween Atoms

- `kTween3dInitialCondition`

  Specifies an initial transform for a 3D tween whose tween type is one of the following: `kTweenType3dCameraData`, `kTweenType3dMatrix`, `kTweenType3dQuaternion`, `kTweenType3dRotate`, `kTweenType3dRotateAboutAxis`, `kTweenType3dRotateAboutAxis`, `kTweenType3dRotateAboutPoint`, `kTweenType3dRotateAboutVector`, `kTweenType3dScale`, or `kTweenType3dTranslate`.

  Its parent atom is a `kTweenEntry` atom.

  A `kTweenEntry` atom can contain only one `kTween3dInitialCondition` atom. The ID of this atom is always `1`. The index of this atom is always `1`.

  This atom is a leaf atom. The data type of its data is as follows:

  - For a `kTweenType3dCameraData` tween, its data type is `TQ3CameraData`.
  - For a `kTweenType3dMatrix` tween, its data type is `TQ3Matrix4x4`.
  - For a `kTweenType3dQuaternion` tween, its data type is `TQ3Quaternion`.
  - For a `kTweenType3dRotate` tween, its data type is `TQ3RotateTransformData`.
  - For a `kTweenType3dRotateAboutAxis` tween, its data type is `TQ3RotateAboutAxisTransformData`.
  - For a `kTweenType3dRotateAboutPoint` tween, its data type is `TQ3RotateAboutPointTransformData`.
  - For a `kTweenType3dRotateAboutVector` tween, its data type is `TQ3PlaneEquation`.
  - For a `kTweenType3dScale` tween, its data type is `TQ3Vector3D`.
  - For a `kTweenType3dTranslate` tween, its data type is `TQ3Vector3D`.

This atom is optional. For each tween type, the default value is the data structure that specifies an identity transform, that is, a transform that does not alter the 3D data.

Interpolation Tween Atoms

- `kTweenOutputMax`

  Specifies the maximum output value of an interpolation tween. If a `kTweenOutputMax` atom is included for an interpolation tween, output values for the tween are scaled to be within the minimum and maximum values. The minimum value is either the value of the `kTweenOutputMin` atom or, if there is no `kTweenOutputMin` atom, `0`. For example, if an interpolation tween has values between 0 and 4, and it has `kTweenOutputMin` and `kTweenOutputMax` atoms with values 1 and 2, respectively, a value of 0 (the minimum value before scaling) is scaled to 1 (the minimum specified by the `kTweenOutputMin` atom), a value of 4 (the maximum value before scaling) is scaled to 2 (the maximum specified by the `kTweenOutputMax` atom), and a value of 3 (three-quarters of the way between the maximum and minimum values before scaling) is scaled to 1.75 (three-quarters of the way between the values of the `kTweenOutputMin` and `kTweenOutputMax` atoms).

  Its parent atom is a `kTweenEntry` atom.

  A `kTweenEntry` atom can contain only on e `kTweenOutputMax` atom. The ID of this atom is always `1`. The index of this atom is always `1`.

  This atom is a leaf atom. The data type of its data is `Fixed`.

  This atom is optional. If it is not included, QuickTime does not scale interpolation tween values.

- `kTweenOutputMin`

  Specifies the minimum output value of an interpolation tween. If both `kTweenOutputMin` and `kTweenOutputMax` atoms are included for an interpolation tween, output values for the tween are scaled to be within the minimum and maximum values. For example, if an interpolation tween has values between 0 and 4, and it has `kTweenOutputMin` and `kTweenOutputMax` atoms with values 1 and 2, respectively, a value of 0 (the minimum value before scaling) is scaled to 1 (the minimum specified by the `kTweenOutputMin` atom), a value of 4 (the maximum value before scaling) is scaled to 2 (the maximum specified by the `kTweenOutputMax` atom), and a value of 3 (three-quarters of the way between the maximum and minimum values before scaling) is scaled to 1.75 (three-quarters of the way between the values of the `kTweenOutputMin` and `kTweenOutputMax` atoms).

If a kTweenOutputMin atom is included but a kTweenOutputMax atom is not, QuickTime does not scale interpolation tween values.

Its parent atom is a kTweenEntry atom.

A kTweenEntry atom can contain only on e kTweenOutputMin atom. The ID of this atom is always 1. The index of this atom is always 1.

This atom is a leaf atom. The data type of its data is Fixed.

This atom is optional. If it is not included but a kTweenOutputMax atom is, the tween component uses 0 as the minimum value for scaling values of an interpolation tween.

- kTweenInterpolationID

  Specifies an interpolation tween to use for a specified kTweenData atom. There can be any number of kTweenInterpolationID atoms for a tween, one for each kTweenData atom to be interpolated.

  Its parent atom is a kTweenEntry atom.

  The index of a kTweenInterpolationID atom specifies when it was added to the kTweenEntry atom; the first added has the index 1, the second 2, and so on. The ID of a kTweenInterpolationID atom must (1) match the atom ID of the kTweenData atom to be interpolated, and (2) be unique among the kTweenInterpolationID atoms contained in the same kTweenEntry atom.

  This atom is a leaf atom. The data type of its data is QTAtomID.

  This atom is required for an interpolation tween.

Region Tween Atoms

- kTweenPictureData

  Contains the data for a QuickDraw picture. Used only by a kTweenTypeQDRegion tween, as described in “Tween Components for QuickDraw 3D” (page 952).

  Its parent atom is a kTweenEntry atom.

  A kTweenEntry atom can contain only on e kTweenPictureData or kTweenRegionData atom. The ID of this atom is always 1. The index of this atom is always 1.

  This atom is a leaf atom. The data type of its data is Picture.

  Either a kTweenPictureData or kTweenRegionData atom is required for a kTweenTypeQDRegion tween.

- `kTweenRegionData`

  Contains the data for a QuickDraw region. Used only by a `kTweenTypeQDRegion` tween, as described in "Tween Components for QuickDraw 3D" (page 952).

  Its parent atom is a `kTweenEntry` atom.

  A `kTweenEntry` atom can contain only on e `kTweenRegionData` or `kTweenPictureData` atom. The ID of this atom is always `1`. The index of this atom is always `1`.

  This atom is a leaf atom. The data type of its data is `Region`.

  Either a `kTweenPictureData` or `kTweenRegionData` atom is required for a `kTweenTypeQDRegion` tween.

Sequence Tween Atoms

- `kTweenSequenceElement`

  Specifies an entry in a tween sequence.

  Its parent is the tween QT atom container (which you specify with the constant `kParentAtomIsContainer`).

  The ID of a `kTweenSequenceElement` atom must be unique among the `kTweenSequenceElement` atoms in the same QT atom container. The index of a `kTweenSequenceElement` atom specifies its order in the sequence; the first entry in the sequence has the index `1`, the second `2`, and so on.

  This atom is a leaf atom. The data type of its data is `TweenSequenceEntryRecord`, a data structure that contains the following fields:

  - `endPercent`, a value of type `Fixed` that specifies the point in the duration of the tween media sample at which the sequence entry ends. This is expressed as a percentage; for example, if the value is 75.0, the sequence entry ends after three-quarters of the total duration of the tween media sample have elapsed. The sequence entry begins after the end of the previous sequence entry or, for the first entry in the sequence, at the beginning of the tween media sample.
  - `tweenAtomID`, a value of type `QTAtomID` that specifies the `kTweenEntry` atom containing the tween for the sequence element. The `kTweenEntry` atom and the `kTweenSequenceElement` atom must both be a child atoms of the same tween QT atom container.
  - `dataAtomID`, a value of type `QTAtomID` that specifies the `kTweenData` atom containing the data for the tween. This atom must be a child atom of the atom specified by the `tweenAtomID` field.

Tween Sequence Entry Record

A tween sequence entry record specifies when a tween in a tween sequence ends. Each tween in a tween sequence begins after the previous tween ends or, for the first tween in the sequence, at the beginning of the tween duration.

Because there can be more than one data set for a tween, the data structure includes a field for the data atom ID as well as the tween atom ID.

```
struct TweenSequenceEntryRecord {
    Fixed                   endPercent;
    QTAtomID                tweenAtomID;
    QTAtomID                dataAtomID;
};
typedef struct TweenSequenceEntryRecord TweenSequenceEntryRecord;
```

Field descriptions

`endPercent`	a value of type `Fixed` that specifies the point in the duration of the tween media sample at which the sequence entry ends.
`tweenAtomID`	a value of type `QTAtomID` that specifies the `kTweenEntry` atom containing the tween for the sequence element.
`dataAtomID`	a value of type `QTAtomID` that specifies the `kTweenData` atom containing the data for the tween.

Component Instance

The component instance for a tween component, `TweenerComponent`, identifies an application's use of a component. For more information about component instances, see the Component Manager chapter of *Inside Macintosh: More Toolbox Essentials*.

```
typedef ComponentInstance TweenerComponent;
```

Tween Record

QuickTime maintains a tween record structure that is provided to your tween component's `TweenDoTween` method. The `TweenRecord` structure is defined as follows.

```
typedef struct TweenRecord TweenRecord;

struct TweenRecord {
    long                    version;
    QTAtomContainer         container;
    QTAtom                  tweenAtom;
    QTAtom                  dataAtom;
    Fixed                   percent;
    TweenerDataUPP          dataProc;
    void *                  private1;
    void *                  private2;
};
```

Field descriptions

`version`	The version number of this structure. This field is initialized to 0.
`container`	The atom container that contains the tween data.
`tweenAtom`	The atom for this tween entry's data in the container.
`percent`	The percentage by which to change the data.
`dataProc`	The procedure the tween component calls to send the tweened value to the receiving track.
`private1`	Reserved.
`private2`	Reserved.

Value Setting Function

The function that you call to send the interpolated value to the receiving track is defined as a universal procedure in systems that support the Macintosh Code Fragment Manager (CFM) or is defined as a data procedure for non-CFM systems:

```
typedef UniversalProcPtr TweenerDataUPP;    /* CFM */

typedef TweenerDataProcPtr TweenerDataUPP;  /* non-CFM */
```

The `TweenerDataUPP` function pointer specifies the function the tween component calls with the value generated by the tween operation. A tween component calls this function from its implementation of the `TweenerDoTween` function.

```
typedef pascal ComponentResult (*TweenerDataProcPtr)(
                        TweenRecord *tr,
                        void *tweenData,
                        long tweenDataSize,
                        long dataDescriptionSeed,
                        Handle dataDescription,
                        ICMCompletionProcRecordPtr asyncCompletionProc,
                        ProcPtr transferProc,
                        void *refCon);
```

`tr`	Contains a pointer to the tween record for the tween operation.
`tweenData`	Contains a pointer to the generated tween value.
`tweenDataSize`	Specifies the size, in bytes, of the tween value.
`dataDescriptionSeed`	Specifies the starting value for the calculation. Every time the content of the `dataDescription` handle changes, this value should be incremented.
`dataDescription`	Specifies a handle containing a description of the tween value passed. For basic types such as integers, the calling tween component should set this parameter to `nil`. For more complex types such as compressed image data, the calling tween component should set this handle to contain a description of the tween value, such as an image description.
`asyncCompletionProc`	Contains a pointer to a completion procedure for asynchronous operations. The calling tween component should set the value of this parameter to `nil`.
`transferProc`	Contains a pointer to a procedure to transfer the data. The calling tween component should set the value of this parameter to `nil`.
`refCon`	Contains a pointer to a reference constant. The calling tween component should set the value of this parameter to `nil`.

DISCUSSION

You call this function by invoking the function specified in the tween record's `dataProc` field.

RESULT CODES

`noErr`	0	No error
`paramErr`	-50	Invalid parameter specified

Tween Component Functions

This section describes the functions that you must provide to implement a tween component. QuickTime calls these functions when it uses your component.

Functions of All Tween Components

TweenerInitialize

The `TweenerInitialize` function is called to initialize your tween component for a single tween operation.

```
extern pascal ComponentResult TweenerInitialize(
                    TweenerComponent tc,
                    QTAtomContainer container,
                    QTAtom tweenAtom,
                    QTAtom dataAtom)
```

`tc`	Specifies the tween component for this operation.
`container`	Specifies the container that holds the atoms specified by the `tweenAtom` and `dataAtom` parameters.
`tweenAtom`	Specifies the atom that contains all parameters for defining this tween. This includes the data atom and any special atoms, such as an atom of type `kTweenRegionData`, that may be necessary.

`dataAtom`	Specifies the atom that contains the values to be tweened. This atom is a child of the atom specified by the `tweenAtom` parameter.

DISCUSSION

This function sets up the tween component when it is first used. In your `TweenerInitialize` function, you can allocate storage and set up any structures that you need for the duration of a tween operation. Although the container that holds the data atom is available during each call to the `TweenerDoTween` function, you can improve the performance of your tween component by extracting the data to be used by the `TweenerDoTween` function in the `TweenerInitialize` function.

The data atom parameter is provided as a convenience; you can also call QT atom container functions to locate the data atom in the container. For more information about the QT atom container functions, see the chapter "Movie Toolbox" in this book.

RESULT CODES

`noErr`	0	No error
`paramErr`	-50	Invalid parameter specified

TweenerReset

The `TweenerReset` function is called to clean up when the tween operation is finished.

```
extern pascal ComponentResult TweenerReset (TweenerComponent tc)
```

`tc`	Specifies the tween component for this operation.

DISCUSSION

This function releases storage allocated by the tween component when the component is no longer being used. The `TweenerReset` function should release any storage allocated by the `TweenerInitialize` function and close or release any other resources used by the component. A tween component may receive a tweener initialize or a close call after being reset.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified

TweenerDoTween

The `TweenerDoTween` function is called to perform a tween operation.

```
extern pascal ComponentResult TweenerDoTween(
                    TweenerComponent tc,
                    TweenRecord *tr)
```

`tc`	Specifies the tween component for this operation.
`tr`	Contains a pointer to the tween record for the tween operation.

DISCUSSION

QuickTime calls this function to interpolate the data used during a tween operation. The tween record contains complete information about the tween operation, including the start and end values for the operation and a percentage that indicates the progress towards completion of the tween sample. For more information about the structure of a tween record, see “Tween Record” (page 969).

Your `TweenerDoTween` function should use the information in the tween record to calculate the tweened value. `TweenerDoTween` should call the data function specified in the tween record, passing it the tweened value. For more information about the data procedure, see “Value Setting Function” (page 970).

RESULT CODES

`noErr`	0	No error
`paramErr`	-50	Invalid parameter specified

Access Key Manager

This chapter describes the Access Key Manager. The Access Key Manager makes it possible to protect data with passwords.

This chapter contains the following major sections:

- "About Access Keys" presents general information about access keys and how applications use them.
- "Using Access Keys" explains how to use access keys in your application.
- "Access Key Reference" describes the constants and functions of the Access Key Manager.

About Access Keys

Access keys make it possible to protect data. It allows an application that supplies data to register a password for the data with QuickTime and allows a user to enter the password to gain access to the data. For example, a codec can protect data it compresses with a password, so that it is available only to someone with the password. Similarly, the creator of a movie can require a password to view the movie.

Although most access keys are character strings, an access key can be of any data type.

System and Application Access Keys

There are two kinds of access keys:

- system access keys
- application access keys.

Data protected by a system access key can be unlocked by any QuickTime caller on the computer. Data protected by an application access key can be unlocked only by QuickTime clients for that application.

System access keys are useful for data that needs to be used by more than one application. When a system access key is registered for data, applications do not have to perform any additional registration to unlock the data. In contrast, application access keys are normally registered by the application in which the data is available, and each application registers the application access keys it uses. For example, a CD-ROM vendor can register the same application access key for all the data on the CD-ROM, which makes the data available to the application on the CD-ROM (such as a game) and inaccessible to all other applications. This prevents browsing of the data by users.

Access Key Types

Access keys are grouped by type. For example, there could be an access key type defined specifically for the Cinepak codec. Grouping access keys lets a QuickTime caller request only those keys that apply to it. This speeds operations involving large numbers of keys which might otherwise interfere with the performance of real-time operations. The functions for using access keys all require an access type.

Using Access Keys

This section explains how to use access keys.

Registering an Access Key

Listing 26-1 illustrates how to register an application access key.

Listing 26-1 Registering an application access key

```
OSErr myErr = 0;
Str255 keyType = doomCDKeyType;
long flags = AccessKeySystemFlag;
```

```
handle keyHdl;

keyHdl = NewHandle (sizeof("keykeykey")-1);
/* put key in handle */
myErr = QTRegisterAccessKey (keyType, flags, keyHdl);
```

Listing 26-2 illustrates how to register a system access key.

Listing 26-2 Registering a system access key

```
OSErr myErr = 0;
Str255 keyType = doomCDKeyType;
long flags = 0;
handle keyHdl;

keyHdl = NewHandle (sizeof("keykeykey")-1);
/* put key in handle */
myErr = QTRegisterAccessKey (keyType, flags, keyHdl);
```

Getting Access Keys

Listing 26-3 illustrates how to get application access keys of a particular type.

Listing 26-3 Getting access keys

```
OSErr myErr = 0;
Str255 keyType = doomCDKeyType;
long flags = 0;
handle keyHdl;

/* handle initialization here */
myErr = QTGetAccessKeys (keyType, flags, keyHdl);
```

Unregistering an Access Key

Listing 26-4 illustrates how to unregister a system access key.

Listing 26-4 Unregistering an access key

```
OSErr myErr = 0;
Str255 keyType = doomCDKeyType;
long flags = AccessKeySystemFlag;
handle keyHdl;

keyHdl = NewHandle (sizeof("keykeykey")-1);
/* put key in handle */
myErr = QTUnRegisterAccessKey (keyType, flags, keyHdl);
```

Access Key Reference

This section describes the constants and functions that are specific to access keys.

Constants

This section provides details about Access Key Manager constants.

```
AccessKeyAtomType            EQU     'acky'
```

This constant specifies the type of atoms in a QT atom container that contain access keys.

```
AccessKeySystemFlag          EQU     $00000001
```

This constant specifies an operation to be performed on a system access key rather than an application access key.

Access Key Manager Functions

The Access Key Manager provides the following functions for registering, getting, and removing access keys.

QTGetAccessKeys

The `QTGetAccessKeys` function returns all the application and system access keys of a specified access key type.

```
pascal OSErr QTGetAccessKeys (
                        Str255 accessKeyType,
                        long flags,
                        QTAtomContainer *keys);
```

`accessKeyType`	The type of access keys to return.
`flags`	Unused; must be set to 0.
`keys`	A pointer to a QT atom container that contains the keys.

DISCUSSION

The QT atom container returned by this function contains atoms of type `kAccessKeyAtomType` at the top level. These atoms contain the keys. You can get the key values by using QT atom functions described in Chapter 1, “Movie Toolbox.” In the QT atom container, application keys, which are more likely to be the ones an application needs, appear before system keys.

If there are no access keys of the specified type, the function returns an empty QT atom container.

When your application is done with the QT atom container, it must dispose of it by calling the `QTDisposeAtomContainer` function.

QTRegisterAccessKey

The `QTRegisterAccessKey` function registers an access key.

```
pascal OSErr QTRegisterAccessKey (
                        Str255 accessKeyType,
                        long flags,
                        Handle accessKey);
```

`accessKeyType`	The access key type of the key to be registered.

`flags`	Flags that specify the operation of this function. To register a system access key, set the `kAccessKeySystemFlag` flag. To register an application access key, set this parameter to 0.
`accessKey`	The key to be registered.

DISCUSSION

Most access keys are strings. A string stored in the `accessKey` handle does not include a trailing zero or leading length byte; to get the length of the string, get the size of the handle.

If the access key has already been registered, no error is returned, and the request is simply ignored.

QTUnregisterAccessKey

The `QTUnregisterAccessKey` function removes a previously registered access key.

```
pascal OSErr QTUnregisterAccessKey (
                        Str255 accessKeyType,
                        long flags,
                        Handle accessKey);
```

`accessKeyType`	The access key type of the key to be removed.
`flags`	Flags that specify the operation of this function. To remove a system access key, set the `kAccessKeySystemFlag` flag. To remove an application access key, set this parameter to 0.
`accessKey`	The key to be removed.

DISCUSSION

Most access keys are strings. A string stored in the `accessKey` handle does not include a trailing zero or a leading length byte.

APPENDIX A

General MIDI Reference

General MIDI Instrument Numbers

Table A-1 General MIDI Instrument Numbers

1	Acoustic Grand Piano	33	Wood Bass
2	Bright Acoustic Piano	34	Electric Bass Fingered
3	Electric Grand Piano	35	Electric Bass Picked
4	Honky-tonk Piano	36	Fretless Bass
5	Rhodes Piano	37	Slap Bass 1
6	Chorused Piano	38	Slap Bass 2
7	Harpsichord	39	Synth Bass 1
8	Clavinet	40	Synth Bass 2
9	Celesta	41	Violin
10	Glockenspiel	42	Viola
11	Music Box	43	Cello
12	Vibraphone	44	Contrabass
13	Marimba	45	Tremolo Strings
14	Xylophone	46	Pizzicato Strings
15	Tubular bells	47	Orchestral Harp
16	Dulcimer	48	Timpani
17	Draw Organ	49	Acoustic String Ensemble 1
18	Percussive Organ	50	Acoustic String Ensemble 2

Table A-1 General MIDI Instrument Numbers

19	Rock Organ	51	Synth Strings 1
20	Church Organ	52	Synth Strings 2
21	Reed Organ	53	Aah Choir
22	Accordion	54	Ooh Choir
23	Harmonica	55	Synvox
24	Tango Accordion	56	Orchestra Hit
25	Acoustic Nylon Guitar	57	Trumpet
26	Acoustic Steel Guitar	58	Trombone
27	Electric Jazz Guitar	59	Tuba
28	Electric clean Guitar	60	Muted Trumpet
29	Electric Guitar muted	61	French Horn
30	Overdriven Guitar	62	Brass Section
31	Distortion Guitar	63	Synth Brass 1
32	Guitar Harmonics	64	Synth Brass 2

65	Soprano Sax	97	Ice Rain
66	Alto Sax	98	Soundtracks
67	Tenor Sax	99	Crystal
68	Baritone Sax	100	Atmosphere
69	Oboe	101	Bright
70	English Horn	102	Goblin
71	Bassoon	103	Echoes
72	Clarinet	104	Space
73	Piccolo	105	Sitar
74	Flute	106	Banjo

75	Recorder	107	Shamisen
76	Pan Flute	108	Koto
77	Bottle blow	109	Kalimba
78	Shakuhachi	110	Bagpipe
79	Whistle	111	Fiddle
80	Ocarina	112	Shanai
81	Square Lead	113	Tinkle bell
82	Saw Lead	114	Agogo
83	Calliope	115	Steel Drums
84	Chiffer	116	Woodblock
85	Synth Lead 5	117	Taiko Drum
86	Synth Lead 6	118	Melodic Tom
87	Synth Lead 7	119	Synth Tom
88	Synth Lead 8	120	Reverse Cymbal
89	Synth Pad 1	121	Guitar Fret Noise
90	Synth Pad 2	122	Breath Noise
91	Synth Pad 3	123	Seashore
92	Synth Pad 4	124	Bird Tweet
93	Synth Pad 5	125	Telephone Ring
94	Synth Pad 6	126	Helicopter
95	Synth Pad 7	127	Applause
96	Synth Pad 8	128	Gunshot

General MIDI Drum Kit Numbers

Table A-2 General MIDI Drum Kit Numbers

35	Acoustic Bass Drum	51	Ride Cymbal 1
36	Bass Drum 1	52	Chinese Cymbal
37	Side Stick	53	Ride Bell
38	Acoustic Snare	54	Tambourine
39	Hand Clap	55	Splash Cymbal
40	Electric Snare	56	Cowbell
41	Lo Floor Tom	57	Crash Cymbal 2
42	Closed Hi Hat	58	Vibraslap
43	Hi Floor Tom	59	Ride Cymbal 2
44	Pedal Hi Hat	60	Hi Bongo
45	Lo Tom Tom	61	Low Bongo
46	Open Hi Hat	62	Mute Hi Conga
47	Low -Mid Tom Tom	63	Open Hi Conga
48	Hi Mid Tom Tom	64	Low Conga
49	Crash Cymbal 1	65	Hi Timbale
50	Hi Tom Tom	66	Lo Timbale

General MIDI Kit Names

Table A-3 General MIDI Kit Names

1	Dry Set
9	Room Set
19	Power Set
25	Electronic Set
33	Jazz Set
41	Brush Set
65-112	User Area
128	Default

QuickTime File Formats

This appendix contains changes and additions to the Motion JPEG and YUV file formats as documented in *QuickTime File Format Specification, May 1996.* as well as new formats supported in QuickTime 2.5 and 3.0.

Motion JPEG

M-JPEG Format A

The following two fields have been added to format A:

Field descriptions

Start of scan offset	Specifies the offset, in bytes, from the start of the field data to the start of the scan marker. This field should never be set to 0.
Start of data offset	Specifies the offset, in bytes, from the start of the field data to the start of the data stream. Typically this immediately follows the start of scan data.

M-JPEG Format B

The following two fields have been added to format B:

Field descriptions

Start of scan offset	Specifies the offset, in bytes, from the start of the field data to the contents of the start of scan data. This field should never be set to 0.
Start of data offset	Specifies the offset, in bytes, from the start of the field data to the start of the data stream. Typically this immediately follows the start of scan data.

YUV

QuickTime 1.6.1 introduced the Component Video codec, which stores data in YUV 4:2:2 format. The compression algorithm is not lossless, but the image quality is extremely high. The compression ratio is 3:2 (or 1.5:1). It does not support frame differencing. This codec is useful for certain video input solutions, such as those included in the Macintosh Quadra and Power Macintosh AV models. The YUV format is also useful as an intermediate storage format if you are applying multiple effects or transitions to an image.

By default, the component video compressor does not appear in the standard compression dialog box. However, it will appear if you hold down the Option key when clicking the compressor list to display the complete list.

Uncompressed YUV2

The YUV2 stream is encoded in a series of four-byte packets. Each packet represents two adjacent pixels on the same scan line. The bytes within each packet are ordered as follows:

```
y0 u y1 v
```

`y0` is the luminance value for the left pixel; `y1` the luminance for the right pixel. `u` and `v` are chromatic values that are shared by both pixels. The conversion into RGB space is represented by the following equations:

$r = 1.402 * v + y + .5$

$g = y - .7143 * v - .3437 * u + .5$

$b = 1.77 * u + y + .5$

The r, g, and b values range from 0 to 255.

QuickTime Image Files

QuickTime image files are intended to provide the most useful container for QuickTime compressed still images. The format uses the same atom-based structure as a QuickTime movie. There are two defined atom types: `'idsc'`, which contains an image description, and `'idat'`, which contains the image data. For a JPEG image, the image description atom contains a QuickTime image description describing the JPEG image's size, resolution, depth, and so on, and the image data atom contains the actual JPEG compressed data. A QuickTime image file can also contain other atoms. For example, it can contain single-fork preview atoms. Because the QuickTime image file is a single fork format, it works well in cross-platform applications. On MacOS systems, QuickTime image files are identified by the file type `'qtif'`. Apple recommends using the filename extension `.QIF` to identify QuickTime image files on other platforms.

DV Files

QuickTime can import, export, decompress, and compress files containing DV data that conforms to the format described in the DV standards document, "Specifications of Consumer-Use Digital VCRs Using 6.3 mm Magnetic Tape" (December, 1994; revised in January, 1996). The type of a Mac OS file containing DV data must be `'dvc!'`; the file extension of a Microsoft Windows file containing DV data must be `.dvc`. The files can contain only the contents of a DV data stream. They cannot contain CIP headers used for IEEE 1394 (FireWire) transport, but they can include header data defined in the DV standards document.

Compressed Movie Resources

Compressed movies, which are described in "Compressed Movie Resources" (page 51), are stored in a compressed movie resource atom container as follows:

- The outermost atom in the movie resource is of type `MovieResourceAtomType`.

  This is the same as for uncompressed movie resources.

- Within the `MovieResourceAtomType` atom is an atom whose ID is `CompressedMovieAID`.

  The `CompressedMovieAID` atom contains two atoms:
  - The atom whose ID is `DataCompressionAtomAID` contains a four-byte code that specifies the decompressor component that must be used to decompress the compressed movie resource.
  - In the atom whose ID is `CompressedMovieDataAID`, the first four bytes of the specify the length of the movie resource data when it is decompressed, in bytes. The remaining bytes are the compressed movie resource.

Component Alias Resources

A component alias is contained in a resource of type `kComponentAliasResourceType`. This resource contains the following data structure:

```
struct ComponentAliasResource {
    ComponentResource       cr;  /* registration parameters */
    ComponentDescription    aliasCD;/* target description */
};
```

The `cr` field is a `ComponentResource` structure that contains information needed to register the component. This information is used in the same way as registration information for ordinary components, with one exception: the specification of the code resource for the component is ignored.

The `aliasCD` field is a `ComponentDescription` structure that specifies the target for the alias. To resolve a component alias, the Component Manager passes the contents of the `aliasCD` field of the `ComponentAliasResource` structure to the `FindNextComponent` function, just as an application would. This field includes all the information that is necessary to specify the target component. The code resource of the target is used in place of the code resource specified in the `cr` field of the `ComponentAliasResource` structure.

A Specification for Storing Floating Point Audio Data

This appendix explains how you can store floating point audio data (IEEE float and double representation) in a QuickTime file.

Note
The word “sample” is used here to describe a single measurement of one channel of an audio signal at a single time. Hence, a sample is represented by one number. Don’t confuse this with the QuickTime “sound sample description” chunk, which describes the format of a collection of audio samples.

The terms “A/D device” and “D/A device” refer to a complete system (i.e., analog circuitry, codec, DMA hardware, kernel software, and library software) for translating between voltage levels at an analog audio jack and numerical values inside the computer (usually stored in memory). An A/D device has an analog input and a D/A device has an analog output. The concepts discussed in this section apply equally as well to other examples, such as digital audio I/O devices. ◆

Mapping Numerical Sample Values with Four Parameters

A device performing audio A/D or D/A conversion must map numerical sample values to voltage levels at its electrical jack in some deterministic way. Similarly, a program that displays an audio waveform on screen must map numerical sample values to screen Y-coordinates in some deterministic way. This section discusses the cases where the numerical sample values are *linearly-related* to the voltage levels or Y-coordinates. Other cases, including compression or companding, then follow trivially.

All possible linear mappings between numerical sample value and voltage or Y-coordinates can be represented with two parameters (i.e., two degrees of freedom). It is more convenient and more common, however, to represent the mappings with four parameters, as explained in the next section.

The A/D – D/A Example

Specifications for an A/D or D/A device will provide you with a voltage level that corresponds to "zero amplitude," and a voltage difference which, when added to or subtracted from the zero amplitude level, produces "full amplitude." The zero amplitude voltage level is typically 0V for DC-coupled outputs. The full amplitude voltage difference depends on the following:

- the device
- the kind of output (i.e., "pro" line levels vs "consumer" line levels vs. speaker levels, etc)
- possibly some settings on the device.

Let's call these two parameters

```
electrical_intercept == the "zero amplitude" voltage level
electrical_slope == the "full amplitude" voltage difference
```

The specifications will return a numerical sample value corresponding to zero amplitude, and a numerical sample difference which, when added to or subtracted from the zero amplitude level, produces full amplitude. Currently, A/D and D/A devices most commonly represent audio with integers, where the numerical values are obvious. The zero amplitude value is usually taken as 0 for two's complement integers and (2^(nbits-1)) for unsigned integers. The full amplitude difference is usually assumed to be (2^(nbits-1))-1. Slightly different assignments, which account for the integer's non-symmetric range about 0, are also sometimes used.

Now let's call these two parameters:

```
numerical_intercept == the "zero amplitude" numerical sample value
numerical_slope == the "full amplitude" numerical sample difference
```

Using all four of these parameters, you can map any numerical value *n* onto the voltage *v* it produces (D/A) or represents (A/D) at the jack:

```
v = electrical_intercept +
    electrical_slope * (n - numerical_intercept) / numerical_slope;
```

These four parameters work together to define a linear mapping between voltage and numerical sample values. In essence, they give meaning to the numbers in the computer. These parameters do *not* impose any constraints on the range or distribution of voltages at the jack or numerical values in the computer. For example:

If numerical_slope==32767 and numerical_intercept==0, it may be entirely possible that the audio data contains no samples with value 32767. There may even be data that falls outside the range of -32767 to 32767.

IMPORTANT

To allay any future confusion: if you are familiar with existing floating point audio file formats, such as Berkeley / IRCAM / CARL, you may be familiar with the "maximum amplitude" tag they contain. This tag is *not* the same as the slope parameter defined here. This tag specifies the highest numerical absolute value of any sample in the file, which may not be the same as the numerical value which maps to full amplitude when played out an electrical jack. ▲

A Waveform Example

The on-screen waveform display is typically what you see in an audio-editing application; time is displayed on a horizontal (X) axis while amplitude is displayed on a vertical (Y) axis. A program that displays an audio waveform on a computer screen has some target Y coordinate where it wants to place zero amplitude data.

Let's call this Y-coordinate screen_intercept. The program also has some differential number of pixels above and below screen_intercept where it wants to place full amplitude data. Let's call this screen_slope.

The program reads audio samples out of memory. As with the A / D – D / A example, there must be a numerical sample value corresponding to "zero amplitude," and a numerical sample difference which when added to or subtracted from the zero amplitude level, produces full amplitude. Let's call these memory_intercept and memory_slope, respectively.

The program then performs exactly the same operation to map numerical sample values *n* to Y-coordinates *y*:

```
y = screen_intercept +
    screen_slope * (n - memory_intercept) / memory_slope;
```

Again, the four parameters define a mapping that allows us to choose Y-coordinates.

Missing Functionality and Limitations

In the two previous examples, you need to map between numerical audio sample values in a computer and some real-world parameter (volts and Y-coordinates). The audio samples in those examples were represented with integers, and so the numerical slope and intercept values to use were fairly obvious.

With floating point data, there is no such obvious assignment. For example, a piece of code may expect or generate -1.0 to 1.0 data, 0.0 to 1.0 data, or -32678.0 to 32767.0 data. This ambiguity actually offers a performance advantage: rather than constantly rescaling output audio data to some standard range after each processing operation, you can often defer the rescale operation until the last minute, when data needs to be played or displayed.

In order for this to work, the slope and intercept parameters described in the previous section must accompany the audio data as it is processed. Each stage of processing "knows" what its algorithm does to the slope and intercept values, and modifies them appropriately.

The issue comes when you store the floating-point audio data into a file. Most current floating point audio file formats do not allow you to store slope and intercept values along with the data. A program that loads such a file and wants to play, display, or further process it must first scan through the entire data set to guess at full amplitude and zero amplitude levels, and even this is not guaranteed to recover the correct values.

This latter scenario is what occurs with floating-point support in NeXT / Sun files. Any attempt to interpret the values in such a soundfile has to be preceded by a long and painful scan step, which renders the file format nearly useless.

The Berkeley / IRCAM / CARL (BICSF) soundfile includes a "maximum amplitude" tag, which must always contain the highest sample absolute value present in the file. The writer of a BICSF file is responsible for computing and storing this maximum amplitude, which in some cases means scanning all of the file's samples, before closing the file.

This scheme is useful in cases where you always want to "normalize" audio samples for playback and display, meaning that you always want the "loudest" part of the audio data to play at the maximum level of which the D/A is capable, and display at the maximum amplitude on a waveform display.

But this is not always what you want. Sometimes, you want the “loudest” part of your audio file to represent some fraction of the maximum level of your D/A device or waveform display. For example, this is often the case in mixdowns. The slope parameter defined here does not have to be the same as (or even greater than) the maximum sample absolute value in the file.

Defining a Superset of the BICSF Functionality

This specification defines a superset of the BICSF functionality. If you want to write a processing module which “normalizes” the data it outputs, you can certainly use the slope parameter in the way you would use the BICSF maximum amplitude tag. But you don’t have to.

The maximum amplitude tag in a BICSF file also helps code deal with clipping issues in floating-point-to-integer conversions more efficiently. This specification defines a separate mechanism to deal with this problem, as explained in the next section.

The BICSF format leaves the intercept parameter undefined. Most, but not all, programs assume that the zero amplitude value is 0.0, and this may sometimes lead to compatibility problems. This specification avoids these problems by making the intercept parameter explicit.

The `slope` and `intercept` parameters fill in a major gap in functionality, and also provide an improvement in performance:

Enhancing Functionality with Slope and Intercept Parameters

The `slope` and `intercept` parameters make it possible to know how the sample values present in a floating point audio file will map onto the zero amplitude value and full amplitude difference of a playback device or a waveform display. This mapping depends *only* on slope and intercept: you can determine the voltage/Y-coordinate corresponding to a numerical sample value without looking at any other samples in the file.

Enhancing Performance

The slope and intercept values make it unnecessary to offset and rescale your data after each stage of processing. You can combine all of these operations into one offset and rescale when you need to play or display the data (and, since you know the slope and the intercept, you may even be able to skip the operations).

Further Performance Improvement for Clipping By Using minclip and maxclip Parameters

When converting floating point values back to integer values for playback or display, clipping is often a performance issue. You want to avoid the extra comparisons and branches if possible.

This specification includes two parameters, `minclip` and `maxclip`, which provide a hint to downstream users of the audio data that no sample value is less than minclip, and no sample value is greater than maxclip. The hint is optional: if you set `minclip` > `maxclip`, that means, in effect, no hint: the samples could have any value. But if you set `minclip` <= `maxclip`, the samples must satisfy the invariants above.

In many cases, it is trivial to determine minclip and maxclip. For example, if you are recording audio to a floating point file and you are capturing 16-bit two's complement integer samples and converting the integers directly into floats, you know that `minclip` is -32768 and `maxclip` is 32767, so you ought to set the parameters.

The `minclip` and `maxclip` parameters are completely independent of `slope` and `intercept`. In the previous example, minclip and maxclip happened to be close to intercept+slope and intercept-slope, but this is not required. For example, if you know that all samples in the above example fall between -1000 and +1000, then go ahead and set `minclip` and `maxclip` to those values. The more constrained you can make the clip (without sacrificing performance), the better.

Now let's say you are playing back a floating point file, so you need to convert floating point data back into 16-bit two's complement integers. The floating point data has a slope of 32767 and an intercept of 0. Do you need to perform clipping operations? The slope and intercept parameters do not tell you this. Their purpose is to define the mapping between the data's zero amplitude value and full amplitude difference and that of an output device or on-screen waveform. From what we know so far, the data could still contain samples with the value -32768.0, or for that matter +1000000000!

You should instead consult `minclip` and `maxclip` to see if clipping is necessary. If the hint is present (`minclip` <= `maxclip`), and `minclip` and `maxclip` are within your desired integral range, you can omit the clipping operations.

This technique does the right thing even if the data is captured with one kind of integer (for example, 16 bit) and played back with another (for example, 24 bit). It also works if the data is synthesized in floating point.

Another issue which enters the clipping picture is how to map effectively the symmetric range of floating point samples onto the non-symmetric range of integers. This is especially important when converting to or from 8-bit integers, where one step is quite audible. Using `slope`, `intercept`, `minclip`, and `maxclip`, you can use any conversion scheme you like, and all the right hints are present to perform the conversion correctly and efficiently.

Using Slope and Intercept

The `slope` parameter must be greater than zero. The `slope` and `intercept` parameters define the meaning of the sample levels in the audio data described by a sound sample description.

A sample in the audio data equal to `intercept` corresponds to the zero amplitude voltage of an analog input or output, and to the zero amplitude Y-coordinate of a graphical waveform display.

A sample in the audio data equal to `intercept + slope` or `intercept - slope` corresponds to full amplitude voltage on an audio input or output, and to full amplitude (i.e., maximum Y coordinate excursion) on a graphical waveform display.

The audio data itself may take any value regardless of the setting of `slope` and `intercept`. These parameters do not define a limit for the audio data, they only define how its values map onto the real-world concepts of zero amplitude and full amplitude. That means it is possible to have a sample value that is more than full amplitude.

A program which modifies data in a QuickTime file by scaling it or offsetting it, but does not intend to change the final amplitude or DC offset of the data when it is played or displayed, should modify `slope` and `offset` accordingly.

Using minclip and maxclip

If `minclip <= maxclip`, then no sample in the audio data has a value less than minclip, and no sample in the audio data has a value greater than `maxclip`. The smallest sample in the audio data may be greater than `minclip` and the largest sample in the audio data may be less than `maxclip`. These two values do not depend on `slope` or `intercept` in any way.

If `minclip > maxclip`, then the sound sample description specifies nothing about the range of sample values present in the audio data.

If a program modifies audio data in a QuickTime file such that `minclip` and `maxclip` no longer bound the range of the audio samples, the program must either update `minclip` and `maxclip`, or set `minclip` greater than `maxclip` to indicate that the limit information is no longer specified.

A P P E N D I X   D

QTAtomContainer-Based Data Structure Descriptions

QTAtomContainer-based data structures are being widely used in QuickTime. This appendix is an attempt at standardizing how the format of these data structure may be described and is documented. The key presented here is used in the QuickTime 3.0 tween documentation in the *QuickTime 3 Reference*.

QTAtomContainer Description Key

```
[(QTAtomFormatName)] =
    atomType_1, id, index
        data
    atomType_n, id, index
        data
```

The atoms may be required or optional:

```
<atomType>optional atom
atomTyperequired atom
```

The atom ID may be a number if it is required to be a constant, or may be a list of valid atom IDs, indicating that multiple atoms of this type are allowed.

`3`	`one atom with id of 3`
`(1..3)`	`three atoms with ids of 1, 2, and 3`
`(1, 5, 7)`	`three atoms with ids of 1, 5, and 7`
`(anyUniqueIDs)`	`multiple atoms each with a unique id`

The atom index may be a 1 if only one atom of this type is allowed, or it may be a range from one to some constant or variable.

```
1               one atom of this type is allowed, index is always 1
(1..3)          three atoms with indicies 1, 2, and 3
(1..numAtoms)   numAtoms atoms with indicies of 1 to numAtoms
```

The data may be leaf data in which its data type is listed inside of brackets [], or may be a nested tree of atoms

```
[theDataType]   leaf data of type theDataType
childAtomsa      nested tree of atoms
```

Nested QT atom format definitions [(`AtomFormatName`)] may appear in a definition.

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 upper bits of a display pixel, which control the blending of video and graphical image data for a video digitizer component.

alternate track A movie **track** that contains alternate data for another track. The Movie Toolbox chooses one track to be used when the movie is played. The choice may be based on such considerations as image quality or localization.

anti-aliasing The process of sampling a signal at more than twice its natural frequency to ensure that **aliasing** artifacts do not occur.

API (Application Programming Interface) The set of function calls, data structures, and other programming elements by which a structure of code (such as a system-level toolbox) can be accesses by other code (such as an application program).

applet A **Java** program or code snippet that can be transmitted over a network and executed on a variety of computers.

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, **sprite,** or other QuickTime data structure. 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.

background color The color of the background behind a sprite or other image.

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.**

bit depth The number of bits used to encode the color of each pixel in a graphics buffer.

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.**

channel component See **sequence grabber channel component.**

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.

clipping The process of defining the boundaries of a graphics area.

clock component A **component** that supplies basic time information to its clients. Clock components have a **component type** value of `'clok'`.

codec A compression / decompression component.

color conversion A stage in **video digitizing** that makes the colors in the image compatible with the destination display.

color ramps Images in which the shading goes from light to dark in smooth increments.

component A software entity, managed by the QuickTime Component Manager, that provides a defined set of services to its clients. Examples include clock components, movie controller components, and image compressor components.

component description record A data structure containing information about a **component,** such as its type and subtype.

component instance A channel of communication between a **component** and its client.

Component Manager A part of QuickTime that lets other software build, install, and manage **components.**

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 event In the **QTMA,** an event that changes the value of a controller for specified parts.

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.

creator signature In the Macintosh file system, a four-character code that identifies the application program to which a file belongs.

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 fork In a Macintosh file, the section that corresponds to a DOS / Windows file.

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.**

desktop sprite A **sprite** that lives in a **sprite world** and can act anywhere on the user's desktop. Desktop sprites are animated by the **Sprite Toolbox.**

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.

dropframe A synchronizing technique that skips timecodes to keep them current with video frames.

duration A time interval. Durations are time values that are interpreted as spans of time, rather than as points in time.

edit list A data structure that arranges a **media** into a time sequence.

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.

effect description A data structure that specifies which component will be used to implement an effect in a movie, and how the component will be configured.

effect track A **modifier track** that applies an effect (such as a wipe or dissolve) to a movie.

file fork A section of a Macintosh file. See **data fork, resource fork.**

file preview A **thumbnail picture** from a movie that is displayed in the Open File dialog box.

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.

general event In the **QTMA,** an event that specifies a synthesizer to be used by subsequent events.

General MIDI component A component built into QuickTime that plays music on any general **MIDI** device.

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.

graphics importer component A QuickTime **component** that opens and displays still images in documents.

graphics mode In **sprite** programming, the **property** of a sprite that determines how it blends with its background.

graphics world A software environment in which a movie track or set of images may be defined before importing them into a movie.

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 In **sprite** programming, one of a sprite's **properties.**

Image Compression Manager (ICM) A QuickTime component that lets other code compress and decompress images and movies.

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.

input map A data structure that describes where to find information about tracks that are targets of a **modifier track.**

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.

ISO Acronym for the International Standards Organization. ISO establishes standards for multimedia data formatting and transmission, such as **JPEG** and **MPEG.**

Java An object-oriented programming language that is widely used to write cross-platform programs, called **applets,** which can be transmitted over a network and run on a variety of platforms. The QuickTime 3.0 **API** is implemented in Java as a set of classes and methods.

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 **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.

knob In the **QTMA,** software that controls the "shape" of musical notes, such as their attack and decay times.

knob event In the **QTMA,** an event that modifies a specific **knob** for specific parts.

layer A mechanism for prioritizing the tracks in a movie or the overlapping of sprites. 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.

marker event In the **QTMA,** an event that specifies the beat or tempo of a series of notes, or the end of a series of events.

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 A defined region of a movie display that can be clipped and filled with another display. 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. Also called a **media structure.**

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.**

MIDI Acronym for Musical Instrument Digital Interface, a standard format for sending instructions to a musical synthesizer.

MIDI component A component built into QuickTime that controls a MIDI synthesizer connected to the computer through a single **MIDI** channel.

modifier track A track in a movie that modifies the data or presentation of other tracks. For example, a **tween** track is a modifier track.

movie A structure 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, its 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.

movie sprite A **sprite** that lives in a **sprite track** and acts in a **movie.**

MPEG-4 A forthcoming **ISO** standard that is based on the QuickTime file format and will support video and audio **streaming.**

music One of the QuickTime media types, in which sequences of sounds and tones are generated.

music control panel A dialog box that lets users select a music synthesizer.

music media handler Part of the QTMA that processes data in the music tracks of QuickTime movies.

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.

note allocator The part of the **QTMA** that plays individual musical notes.

note event In the **QTMA,** an event that specifies the pitch, velocity, and duration of a note.

NTSC See **National Television System Committee.**

Object Metafile In QuickDraw 3D, a file format for storing information about 3D models.

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.

panorama A structure of QuickTime VR data that forms a virtual-world environment within which the user can navigate.

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.**

poster A frame shot from a **movie,** used to represent its content to the user.

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. 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 previews in file dialog boxes to give the user a visual cue about a file's contents. See **file preview.**

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.

property Information about a **sprite** that describes its location or appearance. One sprite property is its **image,** the original bitmapped graphic of the sprite.

QTMA (QuickTime Music Architecture) The part of QuickTime that lets other code create and manipulate music tracks in movies.

QuickDraw The original Mac OS two-dimensional drawing software, used by **QuickTime.**

QuickDraw 3D A cross-platform code library that other code can use to create, configure, and render three-dimensional graphical models. **QuickTime** can import and handle the data objects that it creates.

QuickTime A set of Macintosh system extensions or a Windows dynamic-link library that other code can use to create and manipulate **time-based data.** The current version is QuickTime 3.0.

QuickTime for Java A set of **Java** classes and methods that implements the full **API** of **QuickTime** 3.0.

QuickTime Music Synthesizer A software music synthesizer, built into QuickTime, that plays sounds through the computer's audio hardware.

QuickTime VR A QuickTime media type that lets users interactively explore and examine photorealistic three-dimensional virtual worlds. QuickTime VR data structures are also called **panoramas.**

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.

recording The process of saving captured data in a QuickTime **movie.**

resource In Macintosh programming, an entity in a file or in memory that may contain executable code or a description of a user interface item. Resources are loaded as needed by a resource manager, and are identified by their type and ID number.

resource fork In a Macintosh file, the section that contains **resources.**

rest event In the **QTMA,** an event that specifies the time interval between notes.

sample A single element of a sequence of time-ordered data.

sample format The format of data samples in a track, such as a sprite track.

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 (Systeme Electronique Couleur avec Memoire) 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.

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.**

SMPTE Acronym for Society of Motion Picture and Television Engineers, an organization that sets video and movie technical standards.

Sound Manager A part of the QuickTime built-in software that plays sounds through the computer's audio hardware and also provides tools to manipulate sounds at the system level.

Sound Input Manager A part of the QuickTime built-in software that lets a computer record sounds from a microphone or other audio input device.

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.**

sprite An animated image that is managed by QuickTime. A sprite is defined once and is then animated by commands that change its position or appearance. See **desktop sprite, movie sprite, wired sprite.**

sprite matrix A **property** of a desktop sprite that describes its location and scaling within a **sprite world.**

sprite media handler Part of QuickTime that lets other code install **sprites** in a **sprite track** and manipulate them.

Sprite Toolbox Part of the QuickTime software which provides calls that other software can use to animate **desktop sprites.**

sprite track A movie **track** populated by **movie sprites.**

sprite world A **graphics environment** populated by **desktop sprites.**

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.

standard parameters A dialog box that lets users select and configure video effects.

standard sound A dialog box that lets users select and configure a sound compressor.

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.

streaming Delivery of video or audio data over a network in real time, to support applications such as videophone and video conferencing. See **MPEG-4.**

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 See **SECAM.**

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.**

text channel component A variety of QuickTime **sequence grabber channel component** that uses the services of a **text digitizer** to import text into a **movie.**

text descriptor A QuickTime data structure that stores text formatting commands.

text digitizer component A QuickTime component that obtains text from an external source, such as a file.

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-based data Data that changes or interacts with the user along a time dimension. QuickTime is designed to handle time-based data.

timecode media handler A component that interprets the data in a timecode track.

timecode track A movie track that stores external timing information, such as **SMPTE** timecodes.

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,** which 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 reference A data structure that defines the relation between movie tracks, such as the relation between a **timecode track** and other tracks.

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.

transcoder A **component** that translates data from one compressed form to another without having to decompress it in between.

transformation matrix A 3-by-3 matrix that defines how to map points from one coordinate space into another coordinate space.

tune player The part of the **QTMA** that plays sequences of musical notes.

tween component A component that performs a specific kind of **tweening,** such as path-to-matrix rotation.

tweening A process interpolating new data between given values in conformance to an algorithm. It is an efficient way to expand or smooth a movie's presentation between its actual frames.

tween media handler A component that selects the appropriate **tween component** to perform **tweening.**

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,** such as a modification date or copyright notice.

user data list The collection of **user data** for a QuickTime movie, track, or media. Each element in the user data list is called a **user data item.**

vectors In QuickTime, mathematical descriptions of images that are more compact than bitmaps. The description language is based on Apple Computer's QuickDraw GX technology.

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 bottleneck Places in sequence grabbing code where the grabber will call a client code's **callback function.**

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.

visible In **sprite** programming, the **property** of a sprite that determines whether or not it is visible.

white level The degree of whiteness in an image. It is a common video digitizer setting.

window record A Mac OS data structure that contains a graphics port and information about a window.

window registration In Windows code, the process of associating a window (designated by a `HWND` handle) with QuickTime.

wired sprite A variety of sprite that acts like a button, telling other software when the user has clicked on its image or passed the cursor over it.

Index

This Apple manual was written, edited, and composed on a desktop publishing system using Apple Mac OS computers and Adobe™ FrameMaker software. Line art was created using Adobe™Illustrator and Adobe Photoshop. PostScript™, the page-description language for the LaserWriter, was developed by Adobe Systems Incorporated.

Text type is Palatino® and display type is Helvetica®. Bullets are ITC Zapf Dingbats®. Some elements, such as program listings, are set in Adobe Letter Gothic.

WRITERS
Tom Maremaa and Mitchell Gass

DEVELOPMENTAL EDITOR
Laurel Rezeau

ILLUSTRATOR
David Arrigoni

PRODUCTION EDITOR
Gerri Gray

SPECIAL THANKS TO
Sean Allen, Bob Aron, Jim Batson, Sam Bushell, David van Brink, Greg Chapman, Deeje Cooley, Dan Crow, Mike Dodd, Tom Dowdy, Chris Flick, Brian Friedkin, Steven Gulie, Keith Gurganus, Bill Harris, Michael Hinkson, Peter Hoddie, Jeff Mitchell, Tim Monroe, Guillermo Ortiz, Jim Reekes, George Towner