QuickTime Movie Creation Guide

QuickTime > Movie Creation

2007-01-08

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

Simultaneously published in the United States and Canada.

Contents

Listings

Introduction to QuickTime Movie Creation Guide

This book describes some of the different ways your application can create a new QuickTime movie.

Note: This book replaces six previously separate Apple documents: "Movie Toolbox: Creating Movies," "Sequence Grabber Components," "Sequence Grabber Channel Components," "Sequence Grabber Panel Components," "Text Channel Components," and "Video Digitizer Components."

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

Organization of This Document

This book consists of the following chapters:

- Creating Movies (page 11) shows you how to create a new movie.
- Sequence Grabber Components (page 41) provides an overview of sequence grabber components, channel components, video digitizer components, and panel components (used to provide dialogs).
- Sequence Grabber Component Functions (page 49) describes the functions that are provided by sequence grabber components.
- Sequence Grabber Panel Components (page 77) describes what sequence grabber panel components are, and how they are used.
- Sequence Grabber Channel Components (page 87) describes how to build sequence grabber channel components, also known simply as channel components.
- Using Sequence Grabber Channel Components (page 111) gives an overview of the services your channel component needs to provide.
- Text Channel Components (page 121) describes a type of sequence grabber channel component that captures text for use in QuickTime movies.
- About Video Digitizer Components (page 125) gives you general information about video digitizers in QuickTime.
- Creating Video Digitizer Components (page 131) tells you what support is required from a custom video digitizer component.
- Video Digitizer Component API (page 137) describes the application programming interface for video digitizer components.

See Also

For information about creating a video digitizer component, see *QuickTime Component Creation Guide*.

The following other Apple books cover related aspects of QuickTime programming:

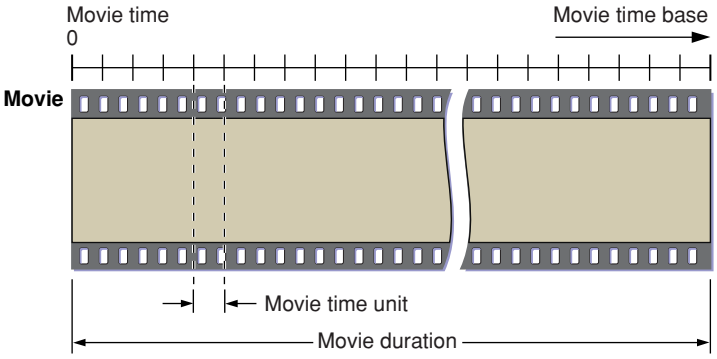
- *QuickTime Overview* gives you the starting information you need to do QuickTime programming.
- *QuickTime Movie Basics* introduces you to some of the basic concepts you need to understand when working with QuickTime movies.
- *QuickTime Guide for Windows* provides information specific to programming for QuickTime on the Windows platform.

Creating Movies

This chapter describes QuickTime movies and shows you how to create a new movie using the QuickTime Movie Toolbox. A sample program is given, detailing the necessary steps: creating and opening a file to hold the movie, creating the tracks and media structures for audio and video, adding sample data, and adding movie resources to the file. Read this section to see a sample program that will step you through the procedure in tutorial fashion.

Movie Structures

QuickTime movies have a time dimension defined by a time scale and a duration, which are specified by a time coordinate system. Figure 1-1 illustrates a movie's time coordinate system. A movie always starts at time 0. The **time scale** defines the unit of measure for the movie's time values. The **duration** specifies how long the movie lasts.



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

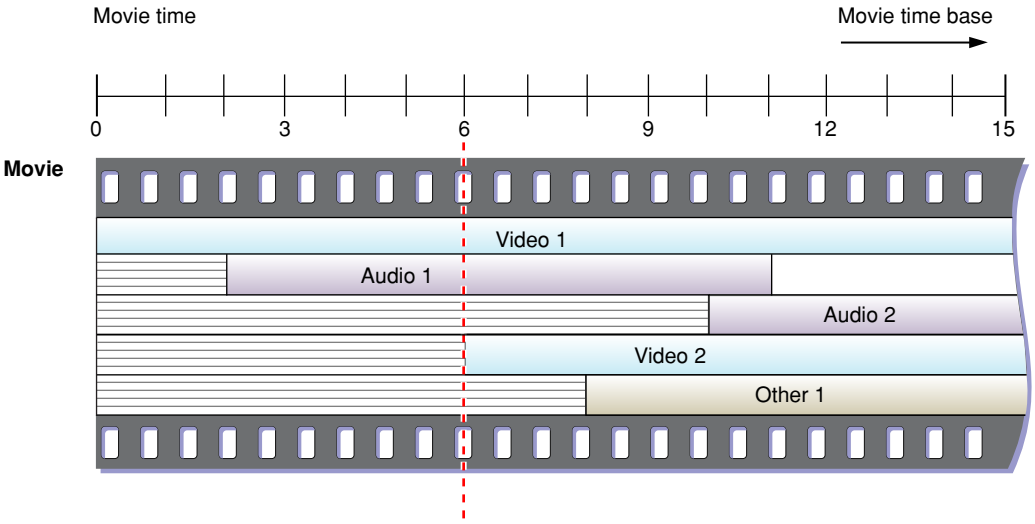
At any given point in time, one or more tracks may or may not be enabled.

Note: Throughout this book, the term *enabled track* denotes a track that may become activated if the movie time intersects the track. An enabled track refers to a media that in turn refers to media data.

However, no single track needs to be enabled during the entire movie. As you move through a movie, you gain access to the data that is described by each of the enabled tracks. Figure 1-2 shows a movie that contains five tracks. The lighter shading in each track represents the time offset between the beginning of the movie and the start of the track's data (this lighter shading corresponds to empty space at the beginning of these

tracks). When the movie's time value is 6, there are three enabled tracks: Video 1 and Audio 1, and Video 2, which is just being enabled. The Other 1 track does not become enabled until the time value reaches 8. The Audio 2 track becomes enabled at time value 10.

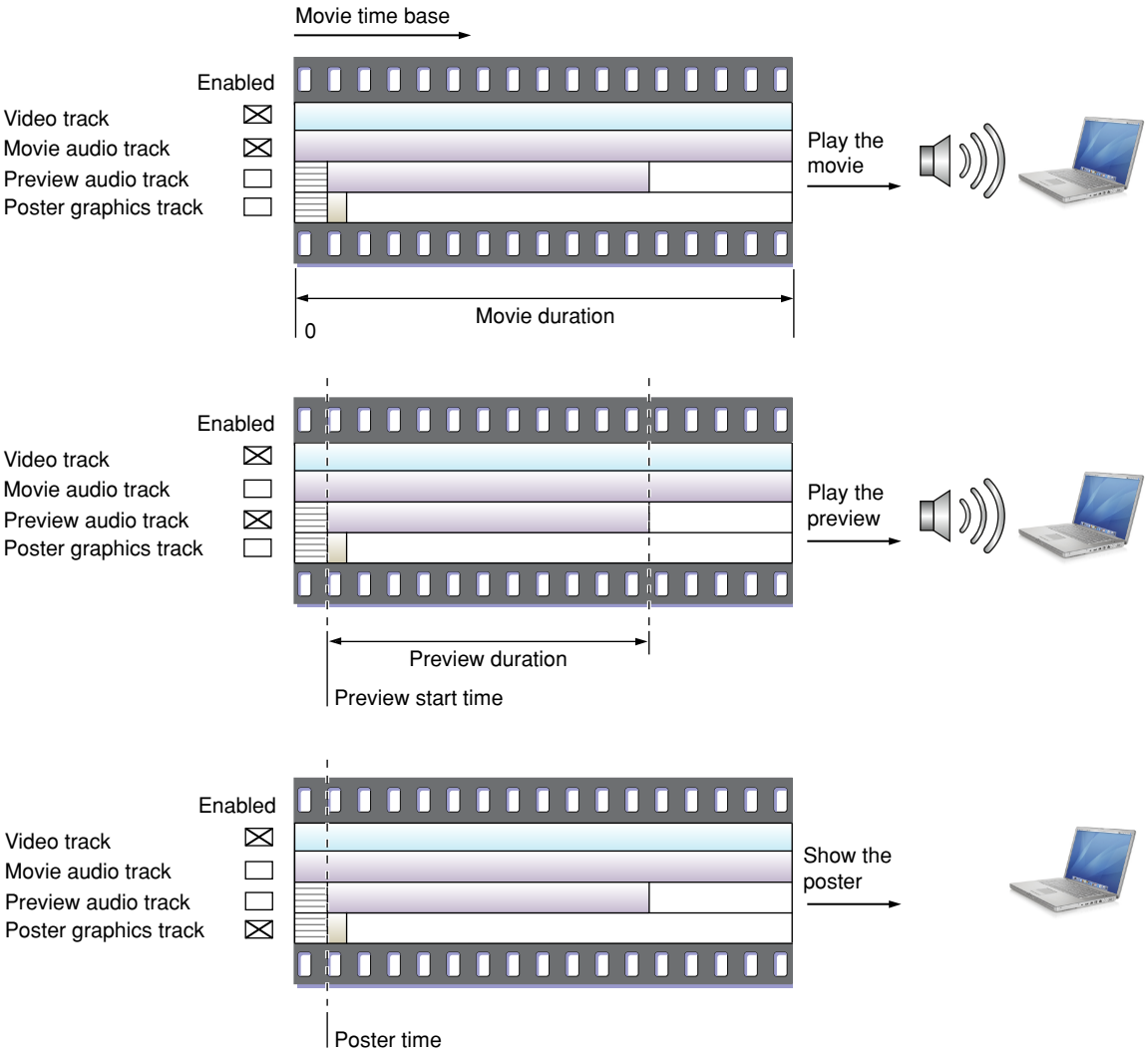
A movie can contain one or more **layers.** Each layer contains one or more tracks that may be related to one another. The Movie Toolbox builds up a movie's visual representation layer by layer. In Figure 1-2, for example, if the images contained in the Video 1 and Video 2 tracks overlap spatially, the user sees the image that is stored in the front layer. You can assign individual tracks to movie layers using Movie Toolbox functions that are described in *QuickTime Movie Internals Guide*.



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

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

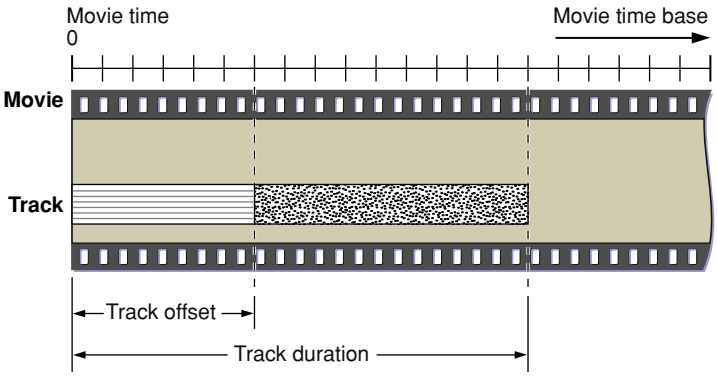
Figure 1-3 shows an example of a movie's tracks. The video track is used for the movie, the preview, and the poster. The movie audio track is used only for the movie. The preview audio track is used only for the preview. The poster graphic track is used only for the poster.



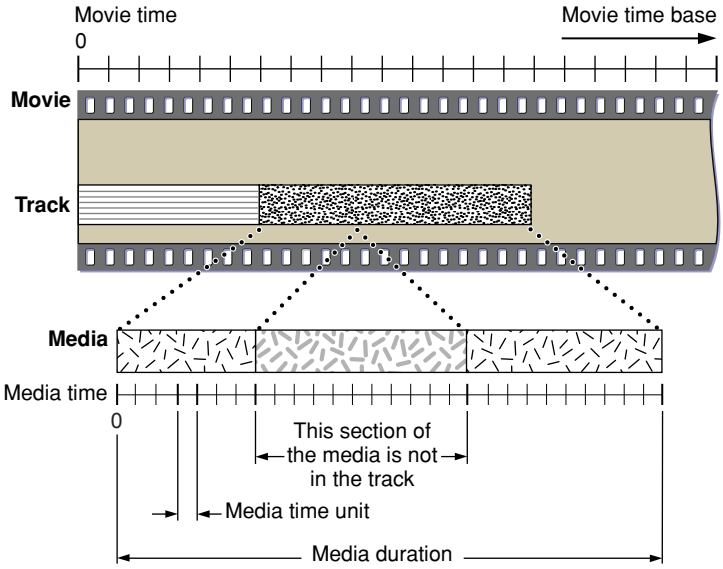
Tracks

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

All of the tracks in a movie use the movie's time coordinate system. That is, the movie's time scale defines the basic time unit for each of the movie's tracks. Each track begins at the beginning of the movie, but the track's data might not begin until some time value other than 0. This intervening time is represented by blank space. In an audio track the blank space translates to silence; in a video track the blank space generates no visual image. Each track has its own duration. This duration need not correspond to the duration of the movie. Movie duration always equals the maximum duration of all the tracks. An example of this is shown in Figure 1-4.



A track is always associated with one media. The media contains control information that refers to the data that constitutes the track. The track contains a list of references that identify portions of the media that are used in the track. In essence, these references are an edit list of the media. Consequently, a track can play the data in its media in any order and any number of times. Figure 1-5 shows how a track maps data from a media into a movie.



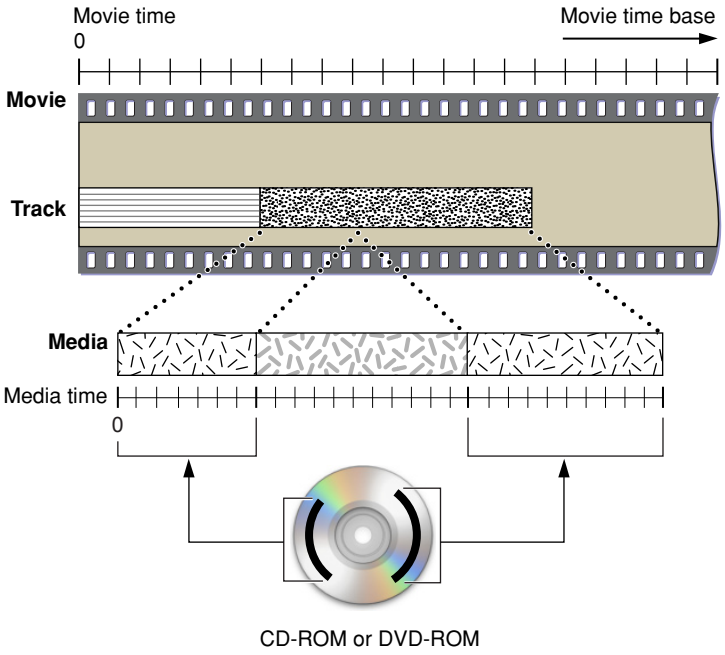
Media Structures

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

Each media has its own time coordinate system, which defines the media's time scale and duration. A media's time coordinate system always starts at time 0, and it is independent of the time coordinate system of the movie that uses its data. Tracks map data from the movie's time coordinate system to the media's time coordinate system. Figure 1-6 shows how tracks perform this mapping.

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

Figure 1-6 shows the final link to the data. The media in the figure references digital video frames on a CD-ROM disc.



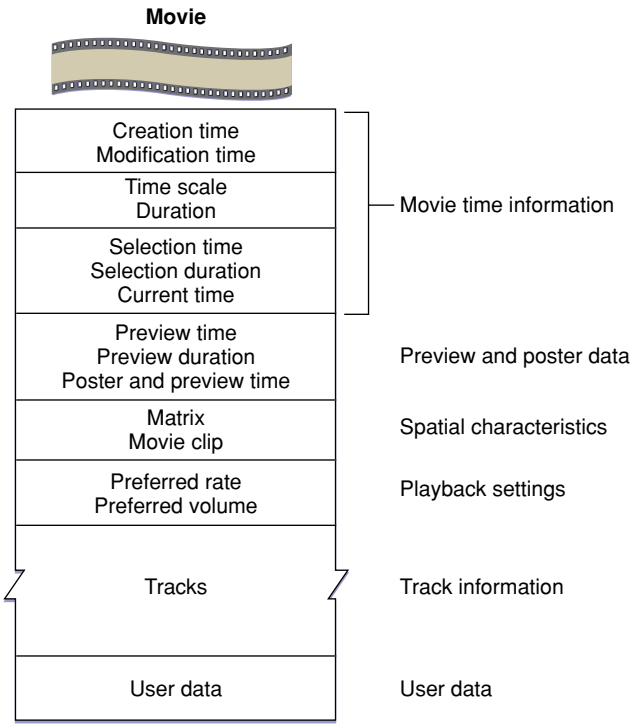
QuickTime Movie Characteristics

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

- Movie Characteristics (page 16) discusses the time, display, and sound characteristics of a QuickTime movie.
- Track Characteristics (page 17) describes the characteristics of a movie track.
- Media Characteristics (page 18) discusses the characteristics of a media.
- Spatial Properties (page 19) describes how the Movie Toolbox displays a movie, including how the data from each media is collected and transformed prior to display.
- The Transformation Matrix (page 24) describes how matrix operations transform visual elements prior to display.
- Audio Properties (page 26) describes how the Movie Toolbox works with a movie's sound tracks.

Movie Characteristics

A QuickTime movie is represented as a private data structure. Your application never works with individual fields in that data structure. Rather, the Movie Toolbox provides functions that allow you to work with a movie's characteristics. Figure 1-7 shows some of the characteristics of a QuickTime movie.



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

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

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

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

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

For each movie currently in use, the Movie Toolbox maintains an **active movie segment.** The active movie segment is the part of the movie that your application is interested in playing. By default, the active movie segment is set to be the entire movie. You may wish to change this to be some segment of the movie; for example, if you wish to play a user's selection repeatedly. By setting the active movie segment, you guarantee that the Movie Toolbox uses no samples from outside of that range while playing the movie.

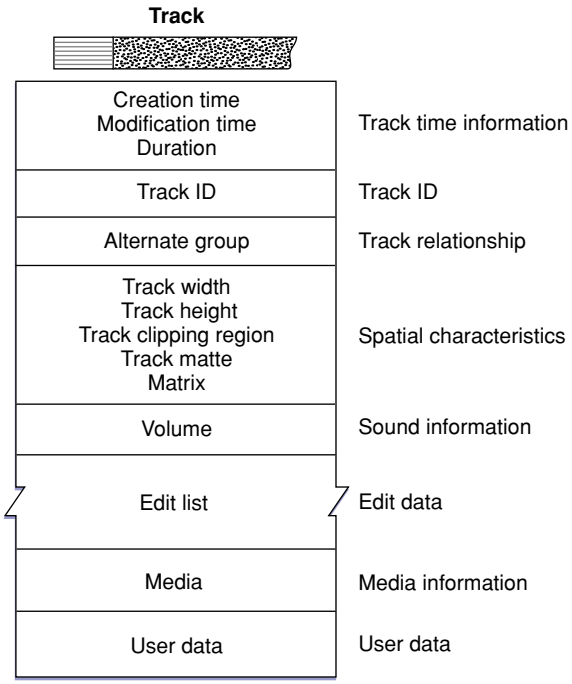
A movie's display characteristics are specified by a number of elements. The movie has a movie clipping region and a 3-by-3 transformation matrix. The Movie Toolbox uses these elements to determine the spatial characteristics of the movie. See Spatial Properties (page 19) for a complete description of these elements and how they are used by the Movie Toolbox.

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

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

Track Characteristics

A QuickTime track is represented as a private data structure. Your application never works with individual fields in that data structure. Rather, the Movie Toolbox provides functions that allow you to work with a track's characteristics. Figure 1-8 shows the characteristics of a QuickTime track.



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

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

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

A track's display characteristics are specified by a number of elements, including track width, track height, a transformation matrix, and a clipping region. See Spatial Properties (page 19) for a complete description of these elements and how they are used by the Movie Toolbox.

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

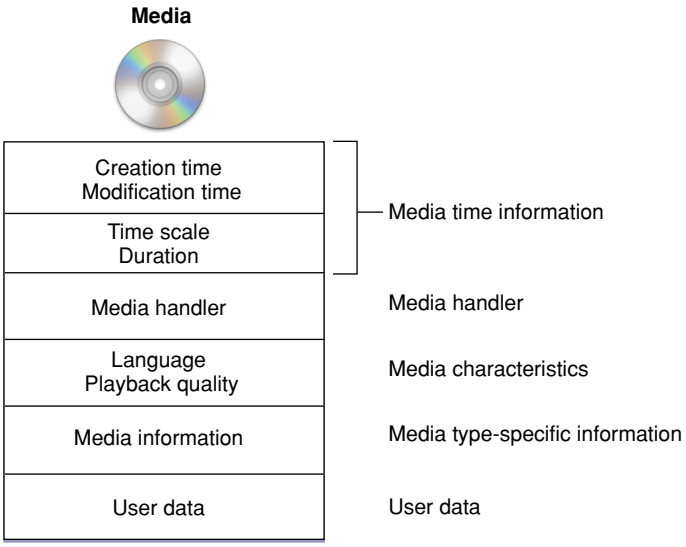
Each track contains a media edit list. The edit list contains entries that define how the track's media is to be used in the movie that contains the track. Each entry in the edit list indicates the starting time and duration of the media segment, along with the playback rate for that segment.

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

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

Media Characteristics

As is the case with movies and tracks, a QuickTime media is represented as a private data structure. Your application never works with individual fields in that data structure. Rather, the Movie Toolbox provides functions that allow you to work with a media's characteristics. Figure 1-9 shows the characteristics of a QuickTime media.



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

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

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

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

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

Spatial Properties

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

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

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

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

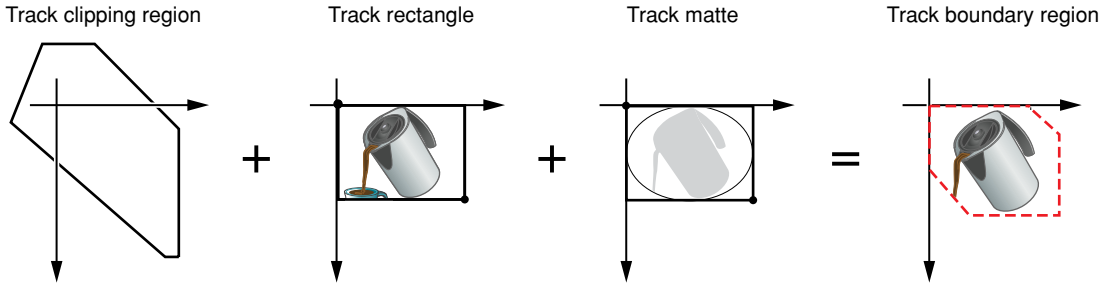
Figure 1-10 provides an overview of the entire process of gathering, transforming, and displaying visual data. Each track defines its own spatial characteristics, which are then interpreted within the context of the movie's spatial characteristics.

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

- the creation of a track rectangle (see Figure 1-11)
- the clipping of a track's image (see Figure 1-12)
- the transformation of a track into the movie coordinate system (see Figure 1-13)
- the clipping of a movie image (see Figure 1-14)
- the transformation of a movie into the display coordinate system (see Figure 1-15)
- the clipping of a movie for final display (see Figure 1-16)

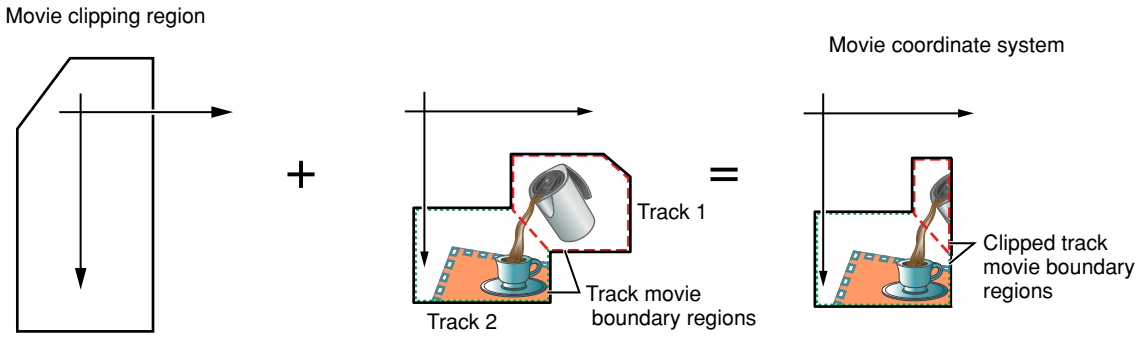
Note: Throughout this book, the term *time coordinate system* denotes QuickTime's time-based system. All other instances of the term *coordinate system* refer to graphic coordinates.

Track coordinate system



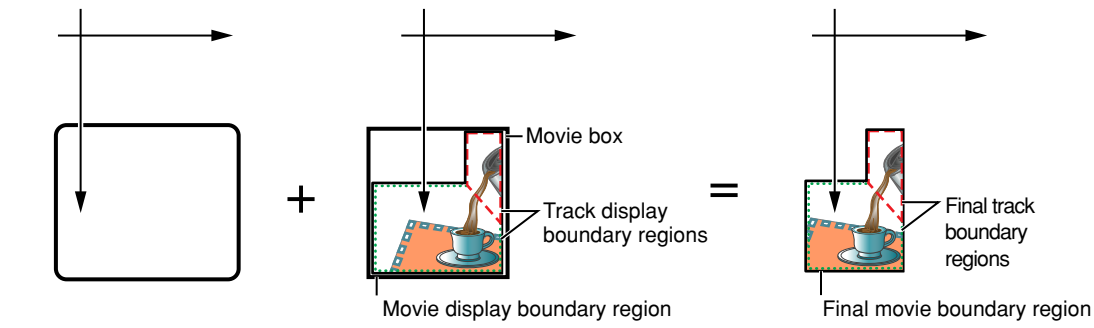
$$\text{Track matrix} = \begin{bmatrix} a & b & u \\ c & d & v \\ x & y & w \end{bmatrix}$$

Movie coordinate system



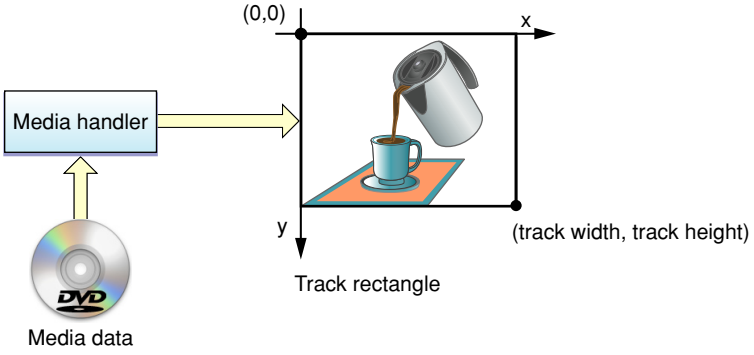
$$\text{Movie matrix} = \begin{bmatrix} a & b & u \\ c & d & v \\ x & y & w \end{bmatrix}$$

Display coordinate system



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

The media handler associated with the track's media is responsible for displaying an image into this rectangle. This process is shown in Figure 1-11.

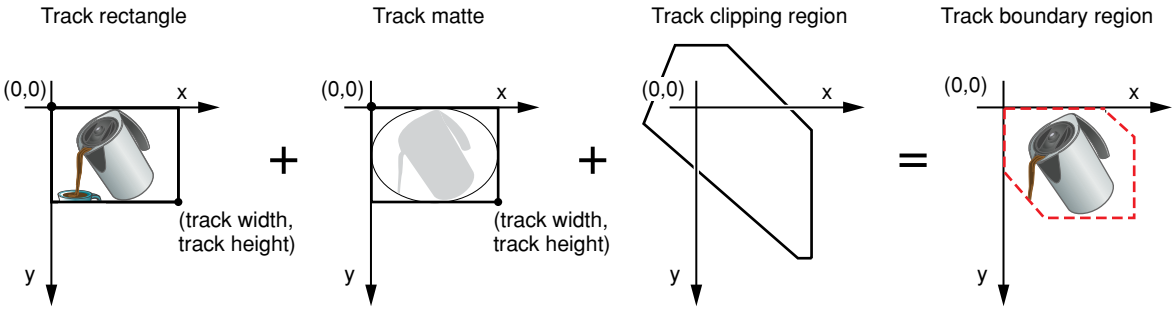


The Movie Toolbox next mattes the image in the track rectangle by applying the track matte and the track clipping region. This does not affect the shape of the image; only the display. Both the track matte and the track clipping region are optional.

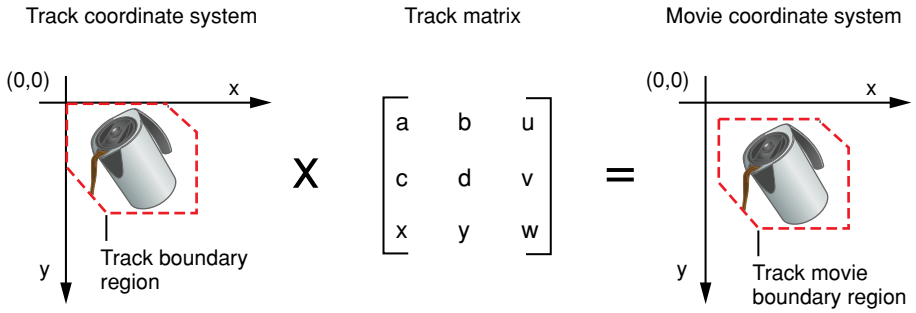
A **track matte** provides a mechanism for mixing images. Mattes contain several bits per pixel and are defined in the track's coordinate system. The matte can be used to perform a deep-mask operation on the image in the track rectangle. The Movie Toolbox displays the weighted average of the track and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

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

This process and its results are shown in Figure 1-12.

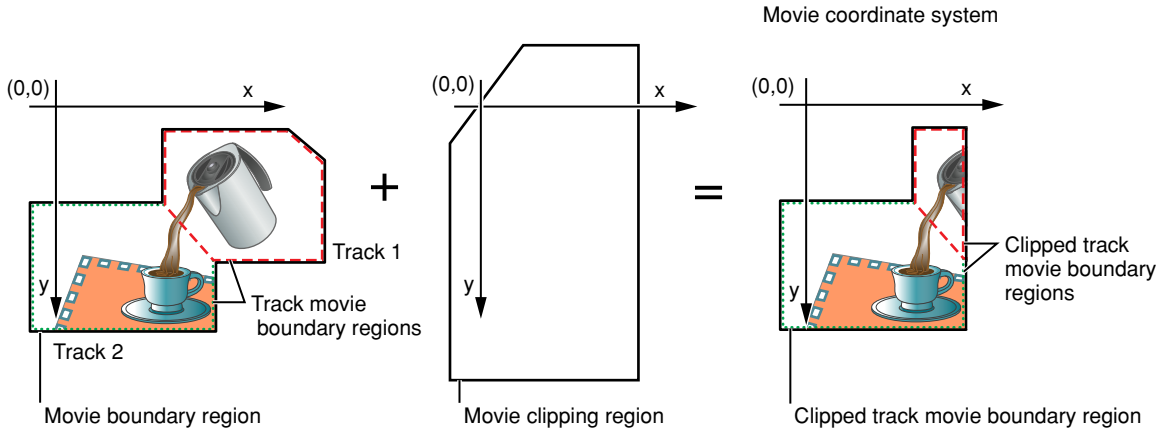


After clipping and matting the track's image, the Movie Toolbox transforms the resulting image into the movie's coordinate system. The Movie Toolbox uses a 3-by-3 transformation matrix to accomplish this operation (see The Transformation Matrix (page 24) for a discussion of matrix operations in the Movie Toolbox). The image inside the track boundary region is transformed by the track's matrix into the movie coordinate system. The resulting area is bounded by the **track movie boundary region.** Figure 1-13 shows the results of this transformation operation.

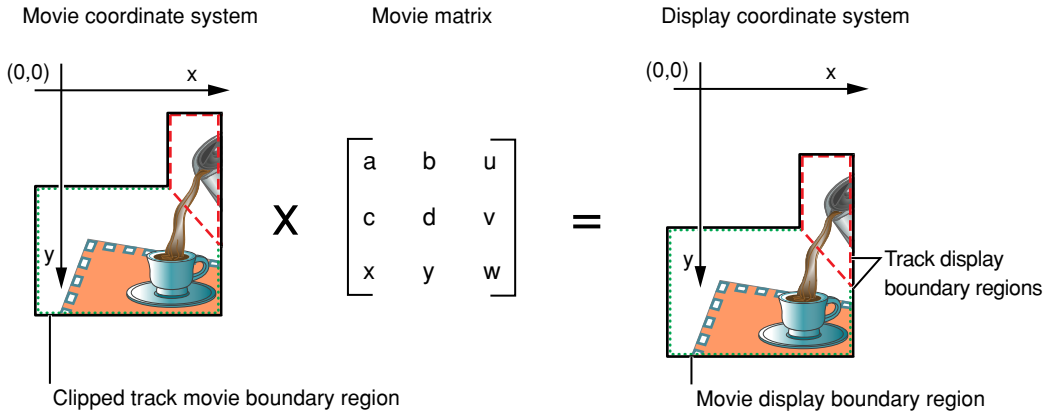


The Movie Toolbox performs this portion of the process for each track in the movie. Once all of the movie's tracks have been processed, the Movie Toolbox proceeds to transform the complete movie image for display.

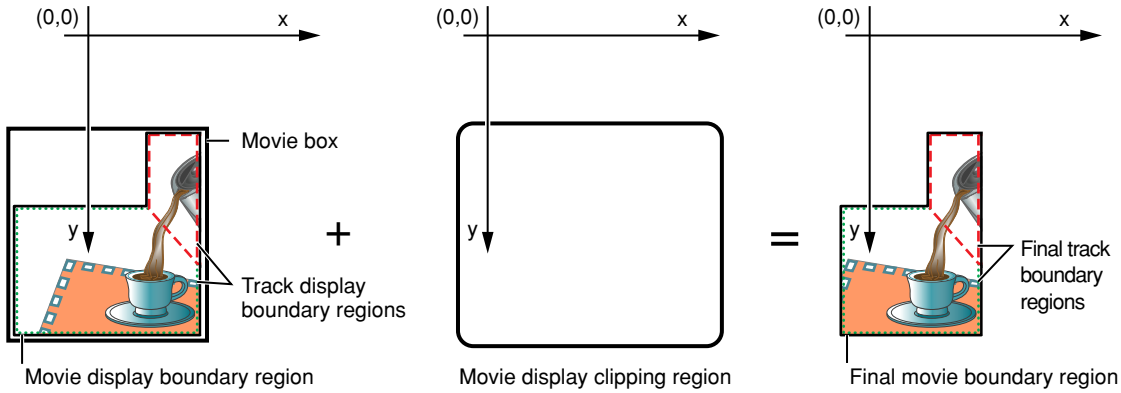
The union of all track movie boundary regions for a movie defines the movie's **movie boundary region.** The Movie Toolbox combines a movie's tracks into this single region where layers are applied. Therefore, tracks in back layers may be partially or completely obscured by tracks in front layers. The Movie Toolbox clips this region to obtain the **clipped movie boundary region.** The movie's **movie clipping region** defines the portion of the movie boundary region that is to be used. Figure 1-14 shows the process by which a movie is clipped and the resulting clipped movie boundary region.



After clipping the movie's image, the Movie Toolbox transforms the resulting image into the display coordinate system. The Movie Toolbox uses a 3-by-3 transformation matrix to accomplish this operation (see The Transformation Matrix (page 24) for a complete discussion of matrix operations in the Movie Toolbox). The image inside the clipped movie boundary region is transformed by the movie's matrix into the display coordinate system. The resulting area is bounded by the movie display boundary region. Figure 1-15 shows the results of this step.



The rectangle that encloses the movie display boundary region is called the **movie box,** as shown in Figure 1-16. You can control the location of a movie's movie box by adjusting the movie's transformation matrix.



Once the movie is in the **display coordinate system** (that is, the QuickDraw graphics world), the Movie Toolbox performs a final clipping operation to generate the image that is displayed. The movie is clipped with the **movie display clipping region.** When a movie is displayed, the Movie Toolbox ignores the graphics port's clipping region; this is why there is a movie display clipping region. Figure 1-16 shows this operation.

The Transformation Matrix

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

The matrix used to accomplish two-dimensional transformations is described mathematically by a 3-by-3 matrix. Figure 1-17 shows a sample 3-by-3 matrix. Note that QuickTime assumes that the values of the matrix elements *u* and *v* are always 0.0, and the value of matrix element *w* is always 1.0.

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

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

x' = ax + cy + t(x)

y' = bx + dy + t(y)

For example, the matrix shown in Figure 1-18 performs no transformation. It is referred to as the **identity matrix.**

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

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

x' = 1x + 0y + 0

y' = 0x + 1y + 0

x' = y and y' = y

In order to move an image by a specified displacement, you perform a translation operation. This operation modifies the x and y coordinates of each point by a specified amount. The matrix shown in Figure 1-19 describes a translation operation.

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

You can stretch or shrink an image by performing a scaling operation. This operation modifies the x and y coordinates by some factor. The magnitude of the x and y factors governs whether the new image is larger or smaller than the original. In addition, by making the x factor negative, you can flip the image about the x-axis; similarly, you can flip the image horizontally, about the y-axis, by making the y factor negative. The matrix shown in Figure 1-20 describes a scaling operation.

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

Finally, you can rotate an image by a specified angle by performing a rotation operation. You specify the magnitude and direction of the rotation by specifying factors for both x and y. The matrix shown in Figure 1-21 rotates an image counterclockwise by an angle *q*.

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

You can combine matrices that define different transformations into a single matrix. The resulting matrix retains the attributes of both transformations. For example, you can both scale and translate an image by defining a matrix similar to that shown in Figure 1-22.

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

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

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

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

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

Audio Properties

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

- Sound Playback (page 27) discusses the playback capabilities of the Movie Toolbox
- Adding Sound to Video (page 27) discusses several issues you should consider when creating movies that contain both sound and video
- Sound Data Formats (page 28) describes the formats the Movie Toolbox uses to store sound information

Sound Playback

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

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

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

Each track in a movie also has a volume setting. A track's volume governs its loudness relative to other tracks in the movie. You can set a track's volume by calling the `SetTrackVolume` function.

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

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

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

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

Adding Sound to Video

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

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

For larger movies, a technique called **interleaving** must be used so that the sound and video data may be alternated in small pieces, and the data can be read off disk as it is needed. Interleaving allows for movies of almost any length with little delay on startup. However, you must tune the storage parameters to avoid a lower video frame rate and breaks in the sound that result when sound data is read from slow storage devices. In general, the Movie Toolbox hides the details of interleaving from your application. The `FlattenMovie` and `FlattenMovieData` functions allow you to enable and disable interleaving when you create a movie. These functions then interact with the appropriate media handler to correctly interleave the sound and video data for your movie.

Sound Data Formats

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

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

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

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

The Movie Toolbox allows you to store information about the sound data in the sound description. See Creating a Sound Description Structure (page 36) for details of the sound description structure. Sample size indicates the number of bits used to encode the amplitude value for each sample. The size of a sample determines the quality of the sound, since more bits can represent more amplitude values. The basic Macintosh sound hardware supports only 8-bit samples, but the Sound Manager also supports 16-bit and 32-bit sample sizes. The Movie Toolbox plays these larger samples on 8-bit Macintosh hardware by converting the samples to 8-bit format before playing them.

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

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

Sample Programs

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

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

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

Main Function

The `CreateWayCoolMovie` program consists of a number of segments, many of which are not included in this sample. Omitted segments deal with general initialization logic and other common aspects of Macintosh programming. The main function, shown in Listing 1-1, shows you how to initialize various parts of the Movie Toolbox and call the `EnterMovies` function.

Listing 1-1 Creating a movie: the main program

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

#include <Memory.h>
#include <Errors.h>
#include <Fonts.h>

#include <QuickDraw.h>
#include <Resources.h>
#include <GestaltEqu.h>
#include <FixMath.h>
```

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

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

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

void InitMovieToolbox (void)
{
        OSErr   err;

        InitGraf (&qd.thePort);
        InitFonts ();
        InitWindows ();
        InitMenus ();
        TEInit ();
        InitDialogs (nil);
        err = EnterMovies ();
        CheckError (err, "\pEnterMovies" );
}

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

Creating and Opening a Movie File

The `CreateMyCoolMovie` function, shown in Listing 1-2, contains the main logic for this program. This function creates and opens a movie file for the new movie. It then establishes a data reference for the movie's data (note that, if your movie's data is stored in the same file as the movie itself, you do not have to create a data reference; set the data reference to 0). This function then calls two other functions, `CreateMyVideoTrack` and `CreateMySoundTrack`, to create the tracks for the new movie. Once the tracks have been created, `CreateMyCoolMovie` adds the new resource to the movie file and closes the movie file.

Listing 1-2 Creating and opening a movie file

```
#define kMyCreatorType 'TVOD'  /* Sample Player's creator type, the
                                  movie player of choice. You can
                                  also use your own creator type. */

#define kPrompt "\pEnter movie file name"

void CreateMyCoolMovie (void)
{
    Point       where = {100,100};
    SFReply     theSFReply;
```

```
    Movie       theMovie = nil;
    FSSpec      mySpec;
    short       resRefNum = 0;
    short       resId = 0;
    OSErr       err = noErr;

    SFPutFile (where, "\pEnter movie file name",
                            "\pMovie File", nil, &theSFReply);
    if (!theSFReply.good) return;
    FSMakeFSSpec(theSFReply.vRefNum, 0,
                                    theSFReply.fName, &mySpec);
    err = CreateMovieFile (&mySpec,
                                'TVOD',
                                smCurrentScript,
                                createMovieFileDeleteCurFile,
                                &resRefNum,
                                &theMovie );
    CheckError(err, "\pCreateMovieFile");
    CreateMyVideoTrack (theMovie);
    CreateMySoundTrack (theMovie);

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

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

The code listing above adds the movie to the resource fork of the file that it creates. It is possible to create a movie file with no resource fork, and to store the movie in the file's data fork.

To create a movie file with no resource fork, pass the `createMovieFileDontCreateResFile` flag when you call `CreateMovieFile`. To store the movie into the file's data fork, call `AddMovieResource` as shown, but pass `kResFileNotOpened` as the `resRefNum` parameter, and pass `movieInDataForkResID` in the `ResID` parameter.

Creating a Video Track in a New Movie

The `CreateMyVideoTrack` function, shown in Listing 1-3, creates a video track in the new movie. This function creates the track and its media by calling the `NewMovieTrack` and `NewTrackMedia` functions, respectively. This function then establishes a media-editing session and adds the movie's data to the media. The bulk of this work is done by the `AddVideoSamplesToMedia` subroutine. Once the data has been added to the media, this function adds the media to the track by calling the Movie Toolbox's `InsertMediaIntoTrack` function.

Listing 1-3 Creating a video track

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

void    CreateMyVideoTrack (Movie theMovie)
{
```

```
    Track           theTrack;
    Media           theMedia;
    OSErr           err = noErr;
    Rect            trackFrame = {0,0,100,320};
    theTrack = NewMovieTrack (theMovie,
                                  FixRatio(trackFrame.right,1),
                                  FixRatio(trackFrame.bottom,1),
                                  kNoVolume);
    CheckError( GetMoviesError(), "\pNewMovieTrack" );

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

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

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

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

Adding Video Samples to a Media

The `AddVideoSamplesToMedia` function, shown in Listing 1-4, creates video data frames, compresses each frame, and adds the frames to the media. This function creates its own video data by calling the `DrawAFrame` function. Note that this function does not temporally compress the image sequence; rather, the function only spatially compresses each frame individually.

Listing 1-4 Adding video samples to a media

```
#define     kSampleDuration             240
        /* video frames last 240 * 1/600th of a second */
#define     kNumVideoFrames             29
#define     kNoOffset                   0
#define     kMgrChoose                  0
#define     kSyncSample                 0
#define     kAddOneVideoSample          1
#define     kPixelDepth                 16
void AddVideoSamplesToMedia (Media theMedia,
                             const Rect *trackFrame)
{
    long                            maxCompressedSize;
    GWorldPtr                       theGWorld = nil;
    long                            curSample;
    Handle                          compressedData = nil;
    Ptr                             compressedDataPtr;
    ImageDescriptionHandle          imageDesc = nil;
    CGrafPtr                        oldPort;
```

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

    LockPixels (theGWorld->portPixMap);

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

    MoveHHi( compressedData );
    HLock( compressedData );
    compressedDataPtr = StripAddress( *compressedData );
    imageDesc = (ImageDescriptionHandle)NewHandle(4);
    CheckError( MemError(), "\pNewHandle" );

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

    for (curSample = 1; curSample < 30; curSample++)
    {
        EraseRect (trackFrame);
        DrawFrame(trackFrame, curSample);
        err = CompressImage (theGWorld->portPixMap,
                                 trackFrame,
                                 codecNormalQuality,
                                 'rle ',
                                 imageDesc,
                                 compressedDataPtr );
        CheckError( err, "\pCompressImage" );

        err = AddMediaSample(theMedia,
                                 compressedData,
                                 0,      /* no offset in data */
                                 (**imageDesc).dataSize,
                                 60,     /* frame duration = 1/10 sec */
                                 (SampleDescriptionHandle)imageDesc,
                                 1,      /* one sample */
                                 0,      /* self-contained samples */
                                 nil);
        CheckError( err, "\pAddMediaSample" );
    }

    SetGWorld (oldPort, oldGDeviceH);
```

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

Creating Video Data for a Movie

The `DrawAFrame` function, shown in Listing 1-5, creates video data for this movie. This function draws a different frame each time it is invoked, based on the sample number, which is passed as a parameter.

Listing 1-5 Creating video data

```
void DrawFrame (const Rect *trackFrame, long curSample)
{
    Str255 numStr;
    ForeColor( redColor );
    PaintRect( trackFrame );
    ForeColor( blueColor );
    NumToString (curSample, numStr);
    MoveTo ( trackFrame->right / 2, trackFrame->bottom / 2);
    TextSize ( trackFrame->bottom / 3);
    DrawString (numStr);
}
```

Creating a Sound Track

The `CreateMySoundTrack` function, shown in Listing 1-6, creates the movie's sound track. This sound track is not synchronized to the video frames of the movie; rather, it is just a separate sound track that accompanies the video data. This function relies upon an `'snd '` resource for its source sound. The `CreateMySoundTrack` function uses the `CreateSoundDescription` function to create the sound description structure for these samples.

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

Once the data has been added to the media, this function adds the media to the track by calling the Movie Toolbox's `InsertMediaIntoTrack` function.

Listing 1-6 Creating a sound track

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

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

    sndHandle = GetResource ('snd ', 128);
    CheckError (ResError(), "\pGetResource" );
    if (sndHandle == nil) return;
    sndDesc = (SoundDescriptionHandle) NewHandle(4);
    CheckError (MemError(), "\pNewHandle" );

    CreateSoundDescription (sndHandle,
                            sndDesc,
                            &sndDataOffset,
                            &numSamples,
                            &sndDataSize );

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

    theMedia = NewTrackMedia (theTrack, SoundMediaType,
                                        FixRound ((**sndDesc).sampleRate),
                                        nil, 0);
    CheckError (GetMoviesError(), "\pNewTrackMedia" );
    err = BeginMediaEdits (theMedia);
    CheckError( err, "\pBeginMediaEdits" );
    err = AddMediaSample(theMedia,
                    sndHandle,
                    sndDataOffset,      /* offset in data */
                    sndDataSize,
                    1,                  /* duration of each sound sample */
                    (SampleDescriptionHandle) sndDesc,
                    numSamples,
                    0,                  /* self-contained samples */
                    nil );
    CheckError( err, "\pAddMediaSample" );

    err = EndMediaEdits (theMedia);
    CheckError( err, "\pEndMediaEdits" );
    err = InsertMediaIntoTrack (theTrack,
                                    0,          /* track start time */
                                    0,          /* media start time */
                                    GetMediaDuration (theMedia),
                                    kFix1);
    CheckError( err, "\pInsertMediaIntoTrack" );
    if (sndDesc != nil) DisposeHandle( (Handle)sndDesc);
}
```

Creating a Sound Description Structure

The `CreateSoundDescription` function, shown in Listing 1-7, creates a sound description structure that correctly describes the sound samples obtained from the `'snd '` resource. This function can handle all the sound data formats that are possible in the sound resource. This function uses the `GetSndHdrOffset` function to locate the sound data in the sound resource.

Listing 1-7 Creating a sound description

```
/* Constant definitions */
#define kMACEBeginningNumberOfBytes 6
#define kMACE31MonoPacketSize 2
#define kMACE31StereoPacketSize 4
#define kMACE61MonoPacketSize 1
#define kMACE61StereoPacketSize 2
void CreateSoundDescription (Handle sndHandle,
                             SoundDescriptionHandlesndDesc,
                             long *sndDataOffset,
                             long *numSamples,
                             long *sndDataSize )
{
    long                        sndHdrOffset = 0;
    long                        sampleDataOffset;
    SoundHeaderPtr              sndHdrPtr = nil;
    long                        numFrames;
    long                        samplesPerFrame;
    long                        bytesPerFrame;
    SignedByte                  sndHState;
    SoundDescriptionPtr         sndDescPtr;

    *sndDataOffset = 0;
    *numSamples = 0;
    *sndDataSize = 0;
    SetHandleSize( (Handle)sndDesc, sizeof(SoundDescription) );
    CheckError(MemError(),"\pSetHandleSize");

    sndHdrOffset = GetSndHdrOffset (sndHandle);
    if (sndHdrOffset == 0) CheckError(-1, "\pGetSndHdrOffset ");
                /* we can use pointers since we don't move memory */
    sndHdrPtr = (SoundHeaderPtr) (*sndHandle + sndHdrOffset);
    sndDescPtr = *sndDesc;

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

    switch (sndHdrPtr->encode)
    {
        case stdSH:
            sndDescPtr->dataFormat = 'raw ';
```

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

    case extSH:
    {
        ExtSoundHeaderPtr           extSndHdrP;
        extSndHdrP = (ExtSoundHeaderPtr)sndHdrPtr;

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

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

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

                default:
                    CheckError(-1, "\pCorrupt sound data" );
                    break;
                }
            }           /* switch cmpSndHdrP->compressionID:*/
            break;      /* of cmpSH: */

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

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

Parsing a Sound Resource

The `GetSndHdrOffset` function, shown in Listing 1-8, parses the specified sound resource and locates the sound data stored in the resource. The `GetSndHdrOffset` function cruises through a specified `'snd '` resource. It locates the sound data, if any, and returns its type, offset, and size into the resource.

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

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

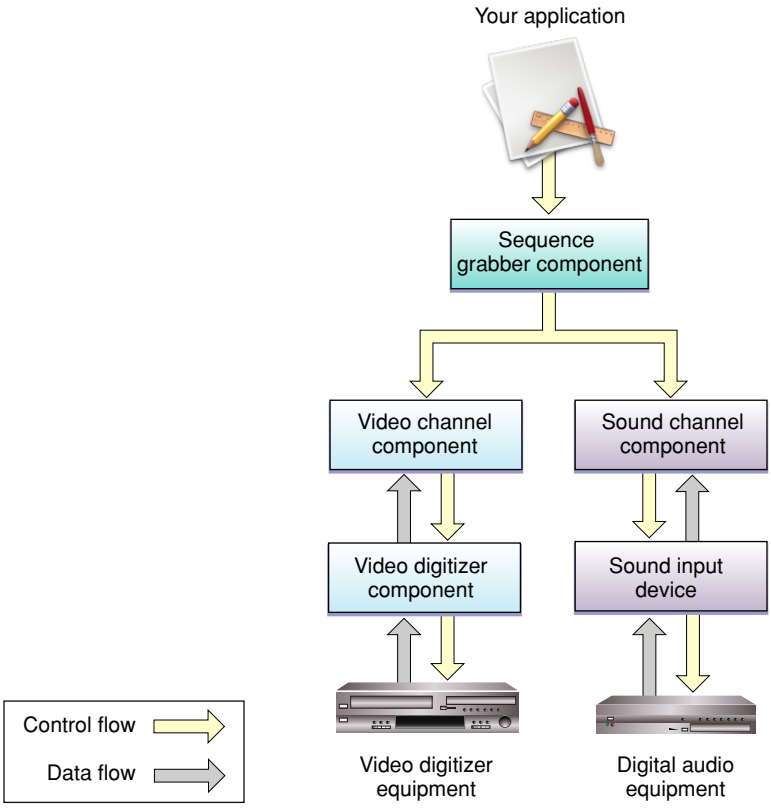
Warning: Do not send the `GetSndHdrOffset` function a `nil` handle; if you do, your system will crash.

Listing 1-8 Parsing a sound resource

```
typedef SndCommand *SndCmdPtr;
typedef struct
{
    short       format;
    short       numSynths;
} Snd1Header, *Snd1HdrPtr, **Snd1HdrHndl;
typedef struct
{
    short       format;
    short       refCount;
} Snd2Header, *Snd2HdrPtr, **Snd2HdrHndl;

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

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

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

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

```
    {
        sndPtr += sizeof(Snd2Header);
    }

    howManyCmds = *(short *)sndPtr;

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

Sequence Grabber Components



Sequence grabber components allow 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. 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.

The sequence grabber component provided by Apple allows applications to capture both audio and video data easily, without concern for the details of how the data is acquired. When capturing video data, this sequence grabber uses a video digitizer component to supply the digitized video images (see About Video Digitizer Components (page 125) for more information about video digitizer components). When working with audio data, Apple's sequence grabber component retrieves its sound data from a sound input device.

Sequence grabber components use sequence grabber channel components (or, simply, channel components) to obtain data from the audio- or video-digitizing equipment. These components isolate the sequence grabber from the details of working with the various types of data that can be collected. The features that a sequence grabber component supplies are dependent on the services provided by sequence grabber channel

components. The channel components, in turn, may use other components to interact with the digitizing equipment. For example, the video channel component supplied by Apple uses a video digitizer component. Figure 2-1 shows the relationship between these components and your application.

Sequence grabber panel components augment the capabilities of sequence grabber components and sequence grabber channel components by allowing sequence grabbers to obtain configuration information from the user for a particular digitizing source. Sequence grabbers present a settings dialog box to the user whenever an application calls the `SGSettingsDialog` function (see Working With Sequence Grabber Settings (page 42) for more information about this sequence grabber function). Applications never call sequence grabber panel components directly; application developers use panel components only by calling the sequence grabber component. See the chapter Sequence Grabber Panel Components (page 77) for more information about the sequence grabber configuration dialog box and the relationships of sequence grabbers, sequence grabber channels, and sequence grabber panels.

If you are developing digitizing equipment and you want to allow applications to use the services of your equipment with a sequence grabber component, you should create an appropriate video digitizer component or sound input device driver.

If you are developing equipment that provides a new type of data to QuickTime, you should develop a new sequence grabber channel component. See Sequence Grabber Channel Components (page 87) for a description of sequence grabber channel components.

Working With Sequence Grabber Settings

Sequence grabber components can work with channel components and panel components to collect configuration settings from the user. The functions discussed in this section allow you to direct the sequence grabber to display its settings dialog box to the user and to work with the configuration of each of the grabber's channels.

Use the `SGSettingsDialog` function to instruct the sequence grabber to display its settings dialog box to the user.

The `SGSetSettings` and `SGGetSettings` functions allow you to retrieve or set the sequence grabber's configuration.

The `SGSetChannelSettings` and `SGGetChannelSettings` functions work with the configuration of an individual channel.

Features of Sequence Grabber Components

Sequence grabber components 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 43) for a complete discussion.

Sequence grabber components use data handler components when writing movie data. This provides greater flexibility, especially when working with special storage devices such as networks.

A 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.

Working with Sequence Grabber Outputs

In order to allow sequence grabber components to capture to more than one data reference at a time, QuickTime supports 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 `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.

Storing Captured Data in Multiple Files

In QuickTime, 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 2-1.

Listing 2-1 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 2-2 shows how this can be accomplished. This example shows how to associate the outputs created in Listing 2-1 with both a sound and a video channel.

Listing 2-2 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 1-3 shows an example of setting a maximum offset of 64 KB for data written to an output.

Listing 2-3 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);
```

Using Sequence Grabber Components

Sequence grabber components are standard components that are managed by the Component Manager.

The sequence grabber component provides functions that give your application precise control over the display of the captured data. This section describes how to use the basic sequence grabber component functions as well as the functions that allow you to configure video and sound channels.

Apple has defined a component type value for sequence grabber components; that type value is `'barg'`. You can use the following constant to specify this type value.

```
#define SeqGrabComponentType 'barg'/* sequence grabber component type */
```

Apple has defined a functional interface for basic sequence grabber components. For information about the functions a sequence grabber component may support, see Sequence Grabber Component Functions (page 49).

You can use the following constants to refer to the request codes for each of the functions that a sequence grabber component may support.

```
enum {
    /* selectors for basic sequence grabber component functions */
    kSGInitializeSelect                         = 0x1;
    kSGSetDataOutputSelect                      = 0x2;
    kSGGetDataOutputSelect                      = 0x3;
    kSGSetGWorldSelect                          = 0x4;
    kSGGetGWorldSelect                          = 0x5;
    kSGNewChannelSelect                         = 0x6;
    kSGDisposeChannelSelect                     = 0x7;
    kSGStartPreviewSelect                       = 0x10;
    kSGStartRecordSelect                        = 0x11;
    kSGIdleSelect                               = 0x12;
    kSGStopSelect                               = 0x13;
    kSGPauseSelect                              = 0x14;
    kSGPrepareSelect                            = 0x15;
    kSGReleaseSelect                            = 0x16;
    kSGGetMovieSelect                           = 0x17;
    kSGSetMaximumRecordTimeSelect               = 0x18;
    kSGGetMaximumRecordTimeSelect               = 0x19;
```

```
    kSGGetStorageSpaceRemainingSelect           = 0x1a;
    kSGGetTimeRemainingSelect                   = 0x1b;
    kSGGrabPictSelect                           = 0x1c;
    kSGGetLastMovieResIDSelect                  = 0x1d;
    kSGSetFlagsSelect                           = 0x1e;
    kSGGetFlagsSelect                           = 0x1f;
    kSGSetDataProcSelect                        = 0x20;
    kSGNewChannelFromComponentSelect            = 0x21;
    kSGDisposeDeviceListSelect                  = 0x22;
    kSGAppendDeviceListToMenuSelect             = 0x23;
    kSGSetSettingsSelect                        = 0x24;
    kSGGetSettingsSelect                        = 0x25;
    kSGGetIndChannelSelect                      = 0x26;
    kSGUpdateSelect                             = 0x27;
    kSGGetPauseSelect                           = 0x28;
    kSGSettingsDialogSelect                     = 0x29;
    kSGGetAlignmentProcSelect                   = 0x2A;
    kSGSetChannelSettingsSelect                 = 0x2B;
    kSGGetChannelSettingsSelect                 = 0x2C;

    /* selectors for common channel configuration functions */
    kSGCSetChannelUsageSelect                   = 0x80;
    kSGCGetChannelUsageSelect                   = 0x81;
    kSGCSetChannelBoundsSelect                  = 0x82;
    kSGCGetChannelBoundsSelect                  = 0x83;
    kSGCSetChannelVolumeSelect                  = 0x84;
    kSGCGetChannelVolumeSelect                  = 0x85;
    kSGCGetChannelInfoSelect                    = 0x86;
    kSGCSetChannelPlayFlagsSelect               = 0x87;
    kSGCGetChannelPlayFlagsSelect               = 0x88;
    kSGCSetChannelMaxFramesSelect               = 0x89;
    kSGCGetChannelMaxFramesSelect               = 0x8a;
    kSGCSetChannelRefConSelect                  = 0x8b;
    kSGCSetChannelClipSelect                    = 0x8C;
    kSGCGetChannelClipSelect                    = 0x8D;
    kSGCGetChannelSampleDescriptionSelect       = 0x8E;
    kSGCGetChannelDeviceListSelect              = 0x8F;
    kSGCSetChannelDeviceSelect                  = 0x90;
    kSGCSetChannelMatrixSelect                  = 0x91;
    kSGCGetChannelMatrixSelect                  = 0x92;
    kSGCGetChannelTimeScaleSelect               = 0x93;

    /* selectors for video channel configuration functions */
    kSGCGetSrcVideoBoundsSelect                 = 0x100;
    kSGCSetVideoRectSelect                      = 0x101;
    kSGCGetVideoRectSelect                      = 0x102;
    kSGCGetVideoCompressorTypeSelect            = 0x103;
    kSGCSetVideoCompressorTypeSelect            = 0x104;
    kSGCSetVideoCompressorSelect                = 0x105;
    kSGCGetVideoCompressorSelect                = 0x106;
    kSGCGetVideoDigitizerComponentSelect        = 0x107;
    kSGCSetVideoDigitizerComponentSelect        = 0x108;
    kSGCVideoDigitizerChangedSelect             = 0x109;
    kSGCSetVideoBottlenecksSelect               = 0x10a;
    kSGCGetVideoBottlenecksSelect               = 0x10b;
    kSGCGrabFrameSelect                         = 0x10c;
    kSGCGrabFrameCompleteSelect                 = 0x10d;
    kSGCDisplayFrameSelect                      = 0x10e;
```

```
    kSGCCompressFrameSelect                     = 0x10f;
    kSGCCompressFrameCompleteSelect             = 0x110;
    kSGCAddFrameSelect                          = 0x111;
    kSGCTransferFrameForCompressSelect          = 0x112;
    kSGCSetCompressBufferSelect                 = 0x113;
    kSGCGetCompressBufferSelect                 = 0x114;
    kSGCGetBufferInfoSelect                     = 0x115;
    kSGCSetUseScreenBufferSelect                = 0x116;
    kSGCGetUseScreenBufferSelect                = 0x117;
    kSGCGrabCompressCompleteSelect              = 0x118;
    kSGCDisplayCompressSelect                   = 0x119;
    kSGCSetFrameRateSelect                      = 0x11A;
    kSGCGetFrameRateSelect                      = 0x11B;

    /* selectors for sound channel configuration functions */
    kSGCSetSoundInputDriverSelect               = 0x100;
    kSGCGetSoundInputDriverSelect               = 0x101;
    kSGCSoundInputDriverChangedSelect           = 0x102;
    kSGCSetSoundRecordChunkSizeSelect           = 0x103;
    kSGCGetSoundRecordChunkSizeSelect           = 0x104;
    kSGCSetSoundInputRateSelect                 = 0x105;
    kSGCGetSoundInputRateSelect                 = 0x106;
    kSGCSetSoundInputParametersSelect           = 0x107;
    kSGCGetSoundInputParametersSelect           = 0x108;

    /* selectors for utility functions provided to channel components */
    kSGWriteMovieData                           = 0x100;
    kSGAddFrameReferenceSelect                  = 0x101;
    kSGGetNextFrameReferenceSelect              = 0x102;
    kSGGetTimeBaseSelect                        = 0x103;
    kSGSortDeviceListSelect                     = 0x104;
    kSGAddMovieDataSelect                       = 0x105;
    kSGChangedSourceSelect                      = 0x106;
};
```

Sequence Grabber Component Functions

This chapter describes the functions that are provided by sequence grabber components. These functions are described from the perspective of an application developer. If you are developing a sequence grabber component, your component must behave as described here.

Configuring Sequence Grabber Components

Sequence grabber components provide a number of functions that allow you to establish the environment for grabbing or previewing digitized data. Before you can start a record or a preview operation, you must initialize the sequence grabber component, establish the channels that will be used, define the display environment for the operation, and determine the optimum screen position for the sequence grabber. In addition, if you are performing a record operation, you must define a destination movie file. The following sequence grabber component functions allow you to perform these tasks:

- You can use the `SGInitialize` function to initialize a sequence grabber component. Before you can call this function, you must establish a connection to the sequence grabber by calling the Component Manager's `OpenDefaultComponent` or `OpenComponent` function.
- The `SGNewChannel` function allows you to create channels for the sequence grabber for an operation. You can use the `SGNewChannelFromComponent` function to create a new channel using a specified channel component. Use the `SGDisposeChannel` function to dispose of those channels that you are no longer using.
- You can use the `SGGetIndChannel` function to retrieve information about the channels that are currently in use by the sequence grabber.
- You can use the `SGSetGWorld` and `SGGetGWorld` functions to establish the display environment for the sequence grabber. These functions affect only those channels that work with data that has visual information.
- The `SGSetDataOutput` and `SGGetDataOutput` functions allow you to identify the movie file that is currently assigned to the sequence grabber. You only use these functions when you are performing a record operation.
- The `SGSetDataProc` function allows you to assign a data function to a channel. The sequence grabber calls your data function whenever it writes movie data to the output file.
- The `SGGetAlignmentProc` function allows you to determine a sequence grabber's optimum screen position to ensure the best performance and appearance.

Controlling Sequence Grabber Components

Sequence grabber components provide a full set of functions that allow your application to control the preview or record operation. You can use these functions to start and stop the operation, to pause data collection, and to retrieve a reference to the movie that is created during a record operation:

- Use the `SGStartPreview` function to start a preview operation. The `SGStartRecord` function lets you start a record operation. The `SGStop` function allows you to stop a sequence grabber component.
- You can instruct the sequence grabber to pause by calling the `SGPause` function. You can determine whether the sequence grabber is paused by calling the `SGGetPause` function.
- You grant processing time to the sequence grabber by calling the `SGIdle` function. Be sure to call this function often during record and preview operations. If your application receives an update event during a record or preview operation, you should call the `SGUpdate` function.
- You can prepare the sequence grabber for an upcoming preview or record operation by calling the `SGPrepare` function. This function also allows the sequence grabber to verify that it can support the parameters you have specified. By verifying the parameters you want to use, you can improve the startup of preview and record operations. Use the `SGRelease` function to release system resources after calling the `SGPrepare` function.
- You can retrieve a reference to the movie created by a record operation by calling the `SGGetMovie` function. You can determine the resource ID value assigned to the last movie resource created by the sequence grabber by calling the `SGGetLastMovieResID` function.
- You can extract a picture from the video source data by calling the `SGGrabPict` function.

Working With Sequence Grabber Characteristics

The characteristics that govern a sequence grabber operation fall into two main categories: those that apply to the sequence grabber component, and those that apply to an individual channel that has been created for the sequence grabber. Sequence grabber components provide a number of functions in each category. The following functions allow you to configure the characteristics of the sequence grabber component. See Working With Channel Characteristics (page 51) for information about functions that apply to a single channel.

- Use the `SGSetMaximumRecordTime` function to limit the duration of a record operation. You can retrieve this time limit by calling the `SGGetMaximumRecordTime` function.
- The `SGSetFlags` function allows you to set control flags that govern an operation. Use the `SGGetFlags` function to retrieve those flags.
- You can obtain information about the progress of a record operation by calling the `SGGetStorageSpaceRemaining` and `SGGetTimeRemaining` functions.
- You can retrieve a reference to the time base used by a sequence grabber component by calling the `SGGetTimeBase` function.

Working With Channel Characteristics

Sequence grabber components use channel components to obtain digitized data from external media. After you create a channel for a sequence grabber component (by calling the `SGNewChannel` function), you must configure that channel before you start a preview or record operation. The sequence grabber component provides a number of functions that allow you to configure the characteristics of a channel component. Several of these functions work on any channel component. This section discusses these general channel configuration functions.

In addition, sequence grabber components provide functions that are specific to the channel type. Apple currently provides three types of channel components: video channel components, sound channel components, and text channel components. See Working With Video Channels (page 53) for information about the sequence grabber configuration functions that work only with video channels. See Working With Sound Channels (page 54) for information about the sequence grabber configuration functions that work only with sound channels. For information about text channels, see Text Channel Components (page 121).

Here are the principal functions that help you work with sequence grabber channel charateristics:

- Use the `SGSetChannelUsage` function to specify how a channel is to be used. You can restrict a channel to use during record or preview operations. In addition, this function allows you to specify whether a channel plays during a record operation. The `SGGetChannelUsage` function enables you to determine a channel's usage.
- The `SGGetChannelInfo` function allows you to determine whether a channel has a visual or an audio representation.
- The `SGSetChannelPlayFlags` function allows you to influence the speed and quality with which the sequence grabber displays captured data. The `SGGetChannelPlayFlags` function lets you determine these flag settings.
- The `SGSetChannelMaxFrames` function establishes a limit on the number of frames that the sequence grabber will capture from a channel. The `SGGetChannelMaxFrames` function allows you to determine that limit.
- The `SGSetChannelBounds` function allows you to set the display boundary rectangle for a channel. Use the `SGGetChannelBounds` function to determine a channel's boundary rectangle.
- The `SGSetChannelVolume` function allows you to control a channel's sound volume. Use the `SGGetChannelVolume` function to determine a channel's volume.
- The `SGSetChannelRefCon` function allows you to set the value of a reference constant that is passed to your callback functions (see Video Channel Callback Functions (page 54) for information about the callback functions that are supported by video channels).
- Use the `SGGetChannelSampleDescription` function to retrieve a channel's sample description. The `SGGetChannelTimeScale` function allows you to obtain the channel's time scale.
- You can modify or retrieve the channel's clipping region by calling the `SGSetChannelClip` or `SGGetChannelClip` function. You can work with a channel's transformation matrix by calling the `SGSetChannelMatrix` and `SGGetChannelMatrix` functions.

Working With Channel Devices

Sequence grabbers provide a number of functions that allow you to determine the device that is attached to a given sequence grabber channel. These devices allow the channel component to control the digitizing equipment. For example, video channels use video digitizer components, and sound channels use sound input drivers. Your application can use these routines to present a list of available devices to the user, allowing the user to select a specific device for each channel.

You may use the `SGGetChannelDeviceList` function to retrieve a list of devices that may be used with a specified channel. You dispose of this device list by calling the `SGDisposeDeviceList` function. You can place one or more device names into a menu by calling the `SGAppendDeviceListToMenu` function. You can use the `SGSetChannelDevice` function to assign a device to a channel.

The Device List Structure

Some of these functions use a device list structure to pass information about one or more channel devices. The `SGDeviceListRecord` data type defines the format of the device list structure.

```
typedef struct SGDeviceListRecord {
    short               count;                  /* count of devices */
    short               selectedIndex;          /* current device */
    long                reserved;               /* set to 0 */
    SGDeviceName        entry[1];               /* device names */
} SGDeviceListRecord, *SGDeviceListPtr, **SGDeviceList;
```

Field	Description
`count`	Indicates the number of devices described by this structure. The value of this field corresponds to the number of entries in the device name array defined by the `entry` field.
`selectedIndex`	Identifies the currently active device. The value of this field corresponds to the appropriate entry in the device name array defined by the `entry` field. Note that this value is 0-relative; that is, the first entry has an index number of 0, the second's value is 1, and so on.
`reserved`	Reserved for Apple. Always set to 0.
`entry`	Contains an array of device name structures. Each structure corresponds to one valid device. The `count` field indicates the number of entries in this array. The `SGDeviceName` data type defines the format of a device name structure.

The Device Name Structure

Device list structures contain an array of device name structures. Each device name structure identifies a single device that may be used by the channel. The `SGDeviceName` data type defines the format of a device name structure.

```
typedef struct SGDeviceName {
```

```
    Str63           name;               /* device name */
    Handle          icon;               /* device icon */
    long            flags;              /* flags */
    long            refCon;             /* set to 0 */
    long            reserved;           /* set to 0 */
} SGDeviceName;
```

Field	Description
`name`	Contains the name of the device. For video digitizer components, this field contains the component's name as specified in the component resource. For sound input drivers, this field contains the driver name.
`icon`	Contains a handle to the device's icon. Some devices may support an icon, which you may choose to present to the user. If the device does not support an icon, or if you choose not to retrieve this information (by setting the `sgDeviceListWithIcons` flag to 0 when you call the `SGGetChannelDeviceList` function), this field is set to `nil`.
`flags`	Reflects the current status of the device. The sequence grabber sets these flags when you retrieve a device list. The `sgDeviceNameFlagDeviceUnavailable` flag is defined. When set to 1, this flag indicates that this device is not currently available.
`refCon`	Reserved for Apple. Always set to 0.
`reserved`	Reserved for Apple. Always set to 0.

Working With Video Channels

Sequence grabber components provide a number of functions that allow you to configure the grabber's video channels. This section describes these configuration functions, which you can use only with video channels. You can determine whether a channel has a visual representation by calling the `SGGetChannelInfo` function. If you want to configure a sound channel, use the functions described in Working With Sound Channels (page 54). If you want to configure general attributes of a channel, use the functions described in Working With Channel Characteristics (page 51).

The `SGGetSrcVideoBounds` function allows you to determine the coordinates of the source video boundary rectangle. This rectangle defines the size of the source video image being captured by the video channel. You can use the `SGSetVideoRect` function to specify a part of the source video boundary rectangle to be captured by the channel. The `SGGetVideoRect` function allows you to determine the active source video rectangle.

Typically, the sequence grabber component uses the Image Compression Manager to compress the video data it captures. You can control many aspects of this image- compression process. Use the `SGSetVideoCompressorType` function to specify the type of image compressor to use. You can determine the type of image compressor currently in use by calling the `SGGetVideoCompressorType` function. You can specify a particular image compressor and set many image-compression parameters by calling the `SGSetVideoCompressor` function. You can determine which image compressor is being used and its parameter settings by calling the `SGGetVideoCompressor` function.

The channel components that supply video data to a sequence grabber component typically work with a video digitizer component. (See About Video Digitizer Components (page 125) for a description of video digitizer components.)

Sequence grabber components provide functions that allow you to work with a channel's video digitizer component. You can use the `SGGetVideoDigitizerComponent` function to determine which video digitizer component is supplying data to a specified channel component. You can set a channel's video digitizer by calling the `SGSetVideoDigitizerComponent` function. If you change any video digitizer settings by calling the video digitizer component directly, you should inform the sequence grabber component by calling the `SGVideoDigitizerChanged` function.

Some video source data may contain unacceptable levels of visual noise or artifacts. One technique for removing this noise is to capture the image and then reduce it in size. During the size reduction process, the noise can be filtered out. Sequence grabber components provide functions that allow you to filter the input video data. The `SGSetCompressBuffer` function sets a filter buffer for a video channel. The `SGGetCompressBuffer` function returns information about your filter buffer.

You can work with a video channel's frame rate by calling the `SGSetFrameRate` and `SGGetFrameRate` functions. You can control whether a channel uses an offscreen buffer by calling the `SGSetUseScreenBuffer` and `SGGetUseScreenBuffer` functions.

Working With Sound Channels

Sequence grabber components provide a number of functions that allow you to configure the grabber's sound channels. This section describes these configuration functions, which you can use only with sound channels. You can determine whether a channel has a sound representation by calling the `SGGetChannelInfo` function. If you want to configure a video channel, use the functions described in Working With Video Channels (page 53). If you want to configure general attributes of a channel, use the functions described in Working With Channel Characteristics (page 51).

Use the `SGSetSoundInputDriver` function to specify a channel's sound input device. You can determine a channel's sound input device by calling the `SGGetSoundInputDriver` function. If you change any attributes of the sound input device, you should notify the sequence grabber component by calling the `SGSoundInputDriverChanged` function. By default, the sequence grabber component uses the sound driver's best settings.

You can control the amount of sound data the sequence grabber works with at one time by calling the `SGSetSoundRecordChunkSize` function. You can determine this value by calling the `SGGetSoundRecordChunkSize` function.

You can control the rate at which the sound channel samples the input data by calling the `SGSetSoundInputRate` function. You can determine the sample rate by calling the `SGGetSoundInputRate` function.

You can control other sound input parameters by using the `SGSetSoundInputParameters` and `SGGetSoundInputParameters` functions.

Video Channel Callback Functions

Sequence grabber components allow you to define a number of callback functions in your application. The sequence grabber calls your functions at specific points in the process of collecting, compressing, and displaying the source video data. By defining callback functions, you can control the process more precisely or customize the operation of the sequence grabber component.

For example, you could use a callback function to draw a frame number on each video frame as it is collected. You could use either a compress callback function or a grab-complete callback function to accomplish this. The compress callback function is called after each frame is collected, in order to compress the frame. The grab-complete callback function is called just before the compress callback function, as soon as the frame has been captured.

The `SGSetVideoBottlenecks` function lets you assign callback functions to a video channel. You can use the `SGGetVideoBottlenecks` function to determine the callback functions that have been assigned to a video channel.

The `SGSetVideoBottlenecks` function accepts a video bottlenecks structure that identifies the callback functions to be assigned to the channel. In addition, the `SGGetVideoBottlenecks` function contains a pointer to this structure.

The video bottlenecks structure is defined by the `VideoBottles` data type as follows:

```
struct VideoBottles {
    short                       procCount;
    GrabProc                    grabProc;
    GrabCompleteProc            grabCompleteProc;
    DisplayProc                 displayProc;
    CompressProc                compressProc;
    CompressCompleteProc        compressCompleteProc;
    AddFrameProc                addFrameProc;
    TransferFrameProc           transferFrameProc;
    GrabCompressCompleteProc    grabCompressCompleteProc;
    DisplayCompressProc         displayCompressProc;
};
typedef struct VideoBottles VideoBottles;
```

Field	Description
`procCount`	Specifies the number of callback functions that may be identified in the structure. Set this field to 9.
`grabProc`	Identifies the grab function. If you are setting a grab function, set this field so that it points to the function's entry point. If you are not setting a grab function, set this field to `nil`.
`grabCompleteProc`	Identifies the grab-complete function. If you are setting a grab-complete function, set this field so that it points to the function's entry point. If you are not setting a grab-complete function, set this field to `nil`.
`displayProc`	Identifies the display function. If you are setting a display function, set this field so that it points to the function's entry point. If you are not setting a display function, set this field to `nil`.
`compressProc`	Identifies the compress function. If you are setting a compress function, set this field so that it points to the function's entry point. If you are not setting a compress function, set this field to `nil`.
`compressCompleteProc`	Identifies the compress-complete function. If you are setting a compress-complete function, set this field so that it points to the function's entry point. If you are not setting a compress-complete function, set this field to `nil`.

Field	Description
`addFrameProc`	Identifies the add-frame function. If you are setting an add-frame function, set this field so that it points to the function's entry point. If you are not setting an add-frame function, set this field to `nil`.
`transferFrameProc`	Identifies the transfer-frame function. If you are setting a transfer-frame function, set this field so that it points to the function's entry point. If you are not setting a transfer-frame function, set this field to `nil`.
`grabCompressComplete-Proc`	Identifies the grab-compress-complete function. If you are setting a grab-compress-complete function, set this field so that it points to the function's entry point. If you are not setting a grab-compress-complete function, set this field to `nil`.
`displayCompressProc`	Identifies the display-compress function. If you are setting a display-compress function, set this field so that it points to the function's entry point. If you are not setting a display-compress function, set this field to `nil`.

The callback functions listed above are described in Application-Defined Functions (page 65).

For information about utility functions that you can use with video channel callback functions, see Utility Functions for Sequence Grabber Channel Components (page 119).

Previewing and Recording Captured Data

You can use sequence grabber components in two ways: to play digitized data for the user or to save captured data in a QuickTime movie. The process of displaying data that is to be captured is called *previewing;* saving captured data in a movie is called *recording.* You can use previewing to allow the user to prepare to make a recording. If you do so, your application can move directly from the preview operation to a record operation, without stopping the process.

Previewing

Previewing captured data involves playing that data for the user as it is captured. For video data, this means displaying the video images on the computer screen. For audio data, this means playing the sound through the computer's sound system.

Here are the steps you must follow to preview captured data:

1. First, you must open a connection to the sequence grabber component. Use the Component Manager's `OpenDefaultComponent` or `OpenComponent` function.

2. Once you have a connection to a sequence grabber component, you must configure the component for the preview operation. Use the `SGSetGWorld` function to set the graphics world in which the preview is to be displayed. Allocate the appropriate channels by calling the `SGNewChannel` function. You must call this function once for each channel to be used by the sequence grabber component. Use the `SGSetChannelUsage` function to specify that each channel is to be used for previewing. You can then

use the appropriate channel configuration functions to prepare the channel for the preview operation. For video channels, use the functions discussed in Working With Video Channels (page 53). For sound channels, use the functions discussed in Working With Sound Channels (page 54).

3. You start the preview operation by calling the `SGStartPreview` function. The sequence grabber component then begins collecting data from the channels that you have created and plays that data appropriately. You can pause and restart the preview by calling the `SGPause` function. Use the `SGStop` function to stop the preview. During the preview operation, be sure to call the `SGIdle` function frequently, so that the sequence grabber and its channels can perform the operation.

4. When you are done previewing, you can start recording or close your connection to the sequence grabber component. When you close the sequence grabber component, it automatically disposes of the channels you created.

Recording

During a record operation, a sequence grabber component collects the data it captures and formats that data into a QuickTime movie. During a record operation, the sequence grabber can also play the captured data for the user. However, the sequence grabber tries to prevent the playback from interfering with the quality of the recording process.

Here are the steps you must follow to record captured data:

1. As with a preview operation, your application must establish a connection to a sequence grabber component. Use the Component Manager's `OpenDefaultComponent` or `OpenComponent` function.

2. Once you have a connection to a sequence grabber component, you must configure the component for the record operation. Use the `SGSetGWorld` function to set the graphics world in which the data is to be displayed. Allocate the appropriate channels by calling the `SGNewChannel` function. You must call this function once for each channel to be used by the sequence grabber component. Use the `SGSetChannelUsage` function to specify that each channel is to be used for recording. At this time, you can specify whether the sequence grabber is to play that channel's data while recording. You can then use the appropriate channel configuration functions to prepare the channel for the record operation. For video channels, use the functions discussed in Working With Video Channels (page 53). For sound channels, use the functions discussed in Working With Sound Channels (page 54).

3. You must specify a movie file for use by the sequence grabber during the record operation. Use the `SGSetDataOutput` function to specify this movie file. This function also allows you to control whether the sequence grabber adds the movie resource to the movie file and whether it replaces existing data or appends the new movie to the file.

4. You can limit the amount of data that is captured during a record operation. The `SGSetMaximumRecordTime` function establishes a time limit for the record operation. The `SGSetChannelMaxFrames` function limits the number of frames of data that the sequence grabber collects from a specific channel.

5. You start the record operation by calling the `SGStartRecord` function. The sequence grabber component then begins collecting data from the channels you have created, stores the data in a QuickTime movie, and, optionally, plays that data appropriately. You can pause and restart the record process by calling the `SGPause` function. During the record operation, be sure to call the `SGIdle` function frequently, so

that the sequence grabber and its channels can perform the operation. Use the `SGStop` function to stop recording. At this time, the sequence grabber saves the movie in your movie file, if you have chosen to do so.

6. When you are done recording, you can go back to previewing or close your connection to the sequence grabber component. When you close the sequence grabber component, it automatically disposes of the channels you created as well as any movies it has created.

Playing Captured Data and Saving It in a QuickTime Movie

This section supplies a sample program that shows how to use a sequence grabber component to preview and record captured data. The program is divided into groups of functions that do the following tasks:

- initialization
- video and sound channel creation
- sequence preview
- capture of sound and video sequences
- drawing over video frames during a capture operation

Initializing a Sequence Grabber Component

Listing 3-1 provides a sample function that creates and initializes a default sequence grabber component for a specified window (using the `OpenDefaultComponent` and `SGInitialize` functions, respectively). It then sets the graphics world of the sequence grabber component to the specified window with the `SGSetGWorld` function. Note that the `CloseComponent` function is called for housekeeping purposes in case the sequence grabber component fails.

Listing 3-1 Initializing a sequence grabber component

```
SeqGrabComponent MakeSequenceGrabber (WindowPtr aWindow)
{
    SeqGrabComponent anSG;
    OSErr err = noErr;

    /* open up the default sequence grabber */
    anSG = OpenDefaultComponent (SeqGrabComponentType, 0);
    if (anSG) {
        /* initialize the default sequence grabber component */
        err = SGInitialize (anSG);
        if (!err) {
        /* set the sequence grabber's graphics world to the
            specified window */
            err = SGSetGWorld (anSG, (CGrafPtr) aWindow, nil);
        }
    }
    if (err && anSG) {
        /* clean up on failure */
        CloseComponent (anSG);
```

```
        anSG = nil;
    }
    return anSG;
}
```

Creating a Sound Channel and a Video Channel

Listing 3-2 supplies a sample function that attempts to create a video channel and a sound channel for the sequence grabber component that was created in Listing 3-1. The boundaries of the video channel are set to the specifications of the `bounds` parameter. The channel's usage is always set to allow previewing. If the value of the `willRecord` parameter is `true`, then the usage of the channel is set to allow recording also.

The `SGNewChannel` function uses the `VideoMediaType` constant to create a video channel and the `SoundMediaType` constant to create a sound channel. The `SGSetChannelBounds` function specifies the boundaries of the video channel. The `SGSetChannelUsage` function specifies whether the video and the sound channels are used for preview or record operations. The `SGDisposeChannel` function cleans up upon failure for each of the channels.

Listing 3-2 Creating a sound channel and a video channel

```
void MakeGrabChannels (SeqGrabComponent anSG,
                               SGChannel *videoChannel,
                               SGChannel *soundChannel,
                               const Rect *bounds, Boolean willRecord)
{
    OSErr err;
    long usage;
    /* figure out the usage */
    usage = seqGrabPreview;                 /* always previewing */
    if (willRecord)
        usage |= seqGrabRecord;             /* sometimes recording */

    /* create a video channel */
    err = SGNewChannel (anSG, VideoMediaType, videoChannel);
    if (!err) {
    /* set boundaries for new video channel */
        err = SGSetChannelBounds (*videoChannel, bounds);

    /* set usage for new video channel */
        if (!err)
            err = SGSetChannelUsage (*videoChannel,
                                     usage | seqGrabPlayDuringRecord);
        if (err) {
            /* clean up on failure */
            SGDisposeChannel (anSG, *videoChannel);
            *videoChannel = nil;
        }
    }

    /* create a sound channel */
    err = SGNewChannel (anSG, SoundMediaType, soundChannel);
    if (!err) {
        /* set usage of new sound channel */
        err = SGSetChannelUsage (*soundChannel, usage);
        if (err) {
```

```
            /* clean up on failure */
            SGDisposeChannel(anSG, *soundChannel);
            *soundChannel = nil;
        }
    }
}
```

Previewing Sound and Video Sequences in a Window

Listing 3-3 shows how to use the sequence grabber component to preview sound and video sequences in a window. Clicking the content area of the window causes the sequence grabber to pause until the mouse button is released.

The Image Compression Manager's `GetBestDeviceRect` function helps you determine the best monitor for the window. The `SGStartPreview` function begins the preview of the sound and video sequences. The `SGIdle` function grants the sequence grabber component the time it needs to preview data. The `SGUpdate` function informs the sequence grabber of the update event. The Window Manager's `BeginUpdate` and `EndUpdate` functions respond to the event. The `SGPause` function instructs the sequence grabber to suspend and resume its preview operation. In this example, it is used to suspend the preview operation while the mouse button is held down. Finally, the `SGStop` function halts the action of the sequence grabber component. The Component Manager's `CloseComponent` function closes the component connection. The Window Manager's `DisposeWindow` function disposes of the window.

Listing 3-3 Previewing sound and video sequences in a window

```
void CheckError(OSErr error, Str255 displayString)
{
    if (error == noErr) return;
    if (displayString[0] > 0)
        DebugStr(displayString);
    ExitToShell();
}
Boolean IsQuickTimeInstalled (void)
{
    short       error;
    long        result;
    error = Gestalt (gestaltQuickTime, &result);
    return (error == noErr);
}
void initialize (void)
{
    OSErr err;

    InitGraf (&qd.thePort);
    InitFonts ();
    InitWindows ();
    InitMenus ();
    TEInit ();
    InitDialogs (nil);
    MaxApplZone();
    if (!IsQuickTimeInstalled())
        CheckError(-1,"\pPlease install QuickTime and try again");
    err = EnterMovies ();
    CheckError(err,"\pUnable to initialize Movie Toolbox");
}
```

```
WindowPtr makeWindow(void)
{
    WindowPtr aWindow;
    Rect windowRect = {0, 0, 120, 160};
    Rect bestRect;
    /* figure out the best monitor for the window */
    GetBestDeviceRect (nil, &bestRect);
    /* put the window in the top left corner of that monitor */
    OffsetRect(&windowRect, bestRect.left + 10, bestRect.top + 50);
    /* create the window */
    aWindow = NewCWindow (nil, &windowRect, "\pGrabber",
                                true, noGrowDocProc, (WindowPtr)-1,
                                true, 0);
    /* and set the port to the new window */
    SetPort(aWindow);
    return aWindow;
}

main (void)
{
    WindowPtr theWindow;
    SeqGrabComponent theSG;
    SGChannel videoChannel, soundChannel;
    Boolean done = false;
    OSErr err;
    initialize();
    theWindow = makeWindow();
    theSG = makeSequenceGrabber(theWindow);
    if (!theSG) return;

    makeGrabChannels(theSG, &videoChannel, &soundChannel,
                        &theWindow->portRect, false);
    if ((videoChannel == nil) && (soundChannel == nil))
        CheckError(-1,"\pNo sound or video available");
    err = SGStartPreview(theSG);
    CheckError(err, "\pCan't start preview");
    while (!done) {
        AlignmentProcRecord alignProc;
        short part;
        WindowPtr whichWindow;
        EventRecord theEvent;

        GetNextEvent(everyEvent, &theEvent);
        switch (theEvent.what) {
            case nullEvent:         /* give the sequence grabber time */
                    err = SGIdle (theSG);
                    if (err) done = true;
                    break;
            case updateEvt:if (theEvent.message == (long)theWindow) {
                                /* inform the sequence grabber of the
                                    update */
                SGUpdate(theSG,((WindowPeek)
                                 theWindow)->updateRgn);
                /* and swallow the update event */
                BeginUpdate(theWindow);
                EndUpdate(theWindow);
            }
```

```
            break;

            case mouseDown:part = FindWindow (theEvent.where,
                                              &whichWindow);
                    if (whichWindow != theWindow) break;
                    switch (part) {
                        case inContent:
                            /* pause until mouse button is
                                released */
                            SGPause (theSG, true);
                            while (StillDown())
                            ;
                            SGPause(theSG, false);
                            break;
                        case inGoAway:
                            done = TrackGoAway (theWindow,
                                                theEvent.where);
                            break;
                        case inDrag:
                            /* pause when dragging window so video
                                doesn't draw in the wrong place */
                            SGPause (theSG, true);
                            SGGetAlignmentProc (theSG, &alignProc);
                            DragAlignedWindow (theWindow,
                                               theEvent.where,
                                               &screenBits.bounds,
                                               nil, &alignProc);
                            SGPause (theSG, false);
                            break;
                        }
                        break;
        }
    }
    /* clean up */
    SGStop (theSG);
    CloseComponent (theSG);
    DisposeWindow (theWindow);
}
```

Capturing Sound and Video Data

Listing 3-4 uses the sequence grabber component to capture ten seconds of sound and video data. It prompts the user for the name of the file to create. The `SGSettingsDialog` function is issued to invoke the default sound and video capture settings dialog boxes. These default dialog boxes allow the user to configure the settings for the capture operations. The `SGSetMaximumRecordTime` function indicates how long the capture operations will last. The `SGStartRecord` function specifies the time at which the capture operations will begin. The `SGIdle` function grants the time needed to confirm the capture operations. Finally, the `SGStop` function and the Window Manager's `DisposeWindow` routine are called in order to complete the capture of the sequences.

Listing 3-4 Capturing sound and video

```
main (void)
{
    WindowPtr theWindow;
```

```
    CGrafPort tempPort;
    SeqGrabComponent theSG;
    SGChannel videoChannel, soundChannel;
    OSErr err;
    initialize();
    theWindow = makeWindow();
    theSG = makeSequenceGrabber(theWindow);
    if (!theSG) return;
    err = setGrabFile(theSG);
    CheckError(err, "\pNo output file");
    makeGrabChannels (theSG, &videoChannel, &soundChannel,
                             &theWindow->portRect, true);
    if ((videoChannel == nil) && (soundChannel == nil))
        CheckError(-1,"\pNo sound or video available");

    if (videoChannel)
        SGSettingsDialog (theSG, videoChannel, 0, nil,
                                 DoTheRightThing, nil, 0);
    if (soundChannel)
        SGSettingsDialog(theSG, soundChannel, 0, nil,
                                DoTheRightThing, nil, 0);
    err = SGSetMaximumRecordTime(theSG, 10 * 60);
    CheckError(err, "\pCan't set max record time");
    err = SGStartRecord (theSG);
    CheckError(err, "\pCan't start record");
    while (!err)
        err = SGIdle (theSG);
    if (err == grabTimeComplete)
        err = noErr;
    CheckError(err, "\pError while recording");
    err = SGStop(theSG);
    CheckError(err, "\pError creating movie");
    CloseComponent(theSG);
    DisposeWindow(theWindow);
}
```

Setting Up the Video Bottleneck Functions

Listing 3-5 shows how to set up the video bottleneck functions of the sequence grabber video channel component. Inside the main event loop in Listing 3-4, you should add the following lines after you call the `SGSetMaximumRecordTime` function.

Listing 3-5 Setting up the video bottleneck functions

```
    if (videoChannel) {
        err = SGSetVideoBottlenecks (videoChannel, &tempPort);
        CheckError(err, "\pCouldn't set video bottlenecks");
    }
```

Drawing Information Over Video Frames During Capture

Listing 3-6 shows how to use the video bottleneck functions of the sequence grabber video channel component to draw the letters “QT” over each video frame as it is captured.

Listing 3-6 Drawing information over video frames during capture

```
pascal ComponentResult myGrabFrameComplete (SGChannel c,
                                            short bufferNum,
                                            Boolean *done,
                                            long refCon)
{
    ComponentResult err;
    /* call the default grab-complete function */
   err = SGGrabFrameComplete (c, bufferNum, done);
    if (*done) {
        /* frame is done */
        CGrafPtr savePort;
        GDHandle saveGD;
        PixMapHandle bufferPM, savePM;
        Rect bufferRect;
        CGrafPtr tempPort = (CGrafPtr)refCon;
        /* set to our temporary port */
        GetGWorld (&savePort, &saveGD);
        SetGWorld (tempPort, nil);
        /* find out about this buffer */
        err = SGGetBufferInfo (c, bufferNum, &bufferPM, &bufferRect,
                                        nil, nil);
        if (!err) {
            /* set up to draw into this buffer */
            savePM = tempPort->portPixMap;
            SetPortPix(bufferPM);
            /* draw some text into the buffer */
            TextMode (srcXor);
            MoveTo (bufferRect.right - 20, bufferRect.bottom - 14);
            DrawString ("\pQT");
            TextMode(srcOr);
            /* restore temporary port */
            SetPortPix (savePM);
        }
        SetGWorld (savePort, saveGD);
    }
    return err;
}

OSErr setupVideoBottlenecks (SGChannel videoChannel, WindowPtr w,
                                    CGrafPtr tempPort)
{
    OSErr err;
    err = SGSetChannelRefCon (videoChannel, (long)tempPort);
    if (!err) {
        VideoBottles vb;
        /* get the current bottlenecks */
        vb.procCount = 9;
        err = SGGetVideoBottlenecks (videoChannel, &vb);
        if (!err) {
            /* add our GrabFrameComplete function */
            vb.grabCompleteProc = myGrabFrameComplete;
            err = SGSetVideoBottlenecks (videoChannel, &vb);
            /* set up the temporary port */
            OpenCPort (tempPort);           /* create a temporary port
                                               for drawing */
            SetRectRgn (tempPort->visRgn, -32000, -32000, 32000,
```

```
                                32000);         /* with a wide open visible
                                                   and clip region . . . */
            CopyRgn (tempPort->visRgn, tempPort->clipRgn);
                                                /* so that you can use it in
                                                   any video buffer */
            PortChanged ((GrafPtr)tempPort);
                                                /* tell QuickDraw about the
                                                   changes */
        }
    }
    return err;
}
```

Application-Defined Functions

This section describes the functions that your application may supply to sequence grabber components.

- Your grab function is used by the sequence grabber component to begin the capture of a frame of video data. Your grab-complete function allows the sequence grabber component to determine whether the current frame-capture operation is complete.
- Your display function enables the sequence grabber component to move a captured video image in an offscreen buffer into the destination buffer for the video channel.
- The sequence grabber component uses your compress function to commence the compression of a captured video image. Your compress-complete function helps the sequence grabber component to find out if the current frame-compression operation is finished.
- Your add-frame function lets the sequence grabber component add a frame to a movie.
- The sequence grabber component uses your transfer-frame function to move a video frame from the capture buffer into the channel's filter buffer.
- You may provide two functions for use with compressed-source devices. Your grab-compress-complete function determines when the current capture and compress operation is complete. Your display-compress function decompresses and displays a frame.
- The sequence grabber calls your data function whenever any of the grabber's channels write data to the movie file.
- If you call the `SGSettingsDialog` function, you must supply a modal-dialog filter function. The interface that your function must provide is discussed on MyModalFilter (page 74).

MyGrabFunction

The sequence grabber component calls your grab function in order to start capturing a frame of video data.

Your grab function must present the following interface:

```
pascal ComponentResult MyGrabFunction (SGChannel c,
                                       short bufferNum,
                                       long refCon);
```

Parameter	Description
c	Specifies the reference that identifies the channel for this operation.
bufferNum	Identifies the buffer for this operation. You can obtain information about this buffer by calling the SGGetBufferInfo function.
refCon	Contains a reference constant value. You can set this value by calling the SGSetChannelRefCon function.

Error constant	Value	Description
cantDoThatInCurrentMode	-9402	Request invalid in current mode

Your grab function can use the sequence grabber component's SGGrabFrame function to support the default behavior.

MyGrabCompleteFunction

The sequence grabber component calls your grab-complete function in order to determine whether the current frame-capture operation is complete. Once a frame has been completely captured, you can modify its contents to suit your needs. For example, you can overlay text onto the video image.

Your function must present the following interface:

```
pascal ComponentResult MyGrabCompleteFunction (SGChannel c,
                                               short bufferNum,
                                               Boolean *done,
                                               long refCon);
```

Parameter	Description
c	Specifies the reference that identifies the channel for this operation.
bufferNum	Identifies the buffer for this operation. You can obtain information about this buffer by calling the SGGetBufferInfo function.
done	Contains a pointer to a Boolean value. Your function sets this Boolean value to indicate whether the frame has been completely captured. Set the Boolean value to true if the capture is complete; set it to false if it is incomplete.
refCon	Contains a reference constant value. You can set this value by calling the SGSetChannelRefCon function.

Error constant	Value	Description
cantDoThatInCurrentMode	-9402	Request invalid in current mode

Your grab-complete function can use the sequence grabber component's `SGGrabFrameComplete` function to support the default behavior.

See Listing 3-6 for a sample grab-complete function. This function draws the letters "QT" over each video frame in the sequence.

MyDisplayFunction

The sequence grabber component calls your display function in order to transfer a captured video image in an offscreen buffer into the destination buffer for the video channel.

Your display function must support the following interface:

```
pascal ComponentResult MyDisplayFunction (SGChannel c,
                                          short bufferNum,
                                          MatrixRecord *mp,
                                          RgnHandle clipRgn,
                                          long refCon);
```

Parameter	Description
`c`	Specifies the reference that identifies the channel for this operation.
`bufferNum`	Identifies the buffer for this operation. You can obtain information about this buffer by calling the `SGGetBufferInfo` function.
`mp`	Contains a pointer to a transformation matrix for the display operation. If there is no matrix for the operation, this parameter is set to `nil`.
`clipRgn`	Contains a handle to the clipping region for the destination image. This region is defined in the destination coordinate system. Apply the clipping region after applying the transformation matrix. If there is no clipping region, this parameter is set to `nil`.
`refCon`	Contains a reference constant value. You can set this value by calling the `SGSetChannelRefCon` function.

Error constant	Value	Description
`cantDoThatInCurrentMode`	-9402	Request invalid in current mode

Your application sets the destination buffer by calling the `SGSetChannelBounds` function.

Your display function can use the sequence grabber component's `SGDisplayFrame` function to support the default behavior.

MyCompressFunction

The sequence grabber component calls your compress function in order to start compressing the captured video image.

Your compress function must support the following interface:

```
pascal ComponentResult MyCompressFunction (SGChannel c,
                                           short bufferNum,
                                           long refCon);
```

Parameter	Description
c	Specifies the reference that identifies the channel for this operation.
bufferNum	Identifies the buffer for this operation. You can obtain information about this buffer by calling the SGGetBufferInfo function.
refCon	Contains a reference constant value. You can set this value by calling the SGSetChannelRefCon function.

Error constant	Value	Description
cantDoThatInCurrentMode	-9402	Request invalid in current mode

Your compress function can use the sequence grabber component's SGCompressFrame function to support the default behavior. This function uses the Image Compression Manager to compress the video image.

MyCompressCompleteFunction

The sequence grabber component calls your compress-complete function in order to determine whether the current frame-compression operation is complete.

Your compress-complete function must support the following interface:

```
pascal ComponentResult MyCompressCompleteFunction (SGChannel c,
                                                   short bufferNum,
                                                   Boolean *done,
                                                   SGCompressInfo *ci,
                                                   long refCon);
```

Parameter	Description
c	Specifies the reference that identifies the channel for this operation.
bufferNum	Identifies the buffer for this operation. You can obtain information about this buffer by calling the SGGetBufferInfo function.
done	Contains a pointer to a Boolean value. Your function sets this Boolean value to indicate whether the frame has been completely compressed. Set the Boolean value to true if the compression is complete; set it to false if it is incomplete.
ci	Contains a pointer to a compression information structure (defined by the SGCompressInfo data type). If the compression is complete, your function must completely format this structure with information that is appropriate to the frame just compressed.

Parameter	Description
refCon	Contains a reference constant value. You can set this value by calling the SGSetChannelRefCon function.

See The Compression Information Structure (page 74), for a description of the SGCompressInfo data type.

Once a frame has been completely compressed, you can add it to the movie. Your compress-complete function can use the sequence grabber component's SGCompressFrameComplete function to support the default behavior.

Error constant	Value	Description
cantDoThatInCurrentMode	-9402	Request invalid in current mode

MyAddFrameFunction

The sequence grabber component calls your add-frame function in order to add a frame to a movie. Your add-frame function must support the following interface:

```
pascal ComponentResult MyAddFrameFunction (SGChannel c,
                                           short bufferNum,
                                           TimeValue atTime,
                                           TimeScale scale,
                                           SGCompressInfo *ci,
                                           long refCon);
```

Parameter	Description
c	Specifies the reference that identifies the channel for this operation.
bufferNum	Identifies the buffer for this operation. You can obtain information about this buffer by calling the SGGetBufferInfo function.
atTime	Specifies the time at which the frame was captured, in the time scale specified by the scale parameter. Your add-frame function can change this value before adding the frame to the movie or before calling the SGAddFrame function. You can determine the duration of a frame by subtracting its capture time from the capture time of the next frame in the sequence.
scale	Specifies the time scale of the movie. You must not change this value.
ci	Contains a pointer to a compression information structure (defined by the SGCompressInfo data type). This structure contains information describing the compression characteristics of the image to be added to the movie.
refCon	Contains a reference constant value. You can set this value by calling the SGSetChannelRefCon function.

See The Compression Information Structure (page 74), for a description of the SGCompressInfo data type.

You can use your add-frame function to modify the contents of the frame before it is added to the movie. This can be useful if you want to place frame numbers onto frames you are recording.

Error constant	Value	Description
`cantDoThatInCurrentMode`	-9402	Request invalid in current mode

Your add-frame function can use the sequence grabber component's `SGAddFrame` function to support the default behavior.

MyTransferFrameFunction

The sequence grabber component calls your transfer-frame function in order to move a video frame from the capture buffer into the channel's filter buffer.

Your transfer-frame function must support the following interface:

```
pascal ComponentResult MyTransferFrameFunction (SGChannel c,
                                        short bufferNum,
                                        MatrixRecord *mp,
                                        RgnHandle clipRgn,
                                        long refCon);
```

Parameter	Description
`c`	Specifies the reference that identifies the channel for this operation.
`bufferNum`	Identifies the buffer for this operation. You can obtain information about this buffer by calling the `SGGetBufferInfo` function.
`mp`	Contains a pointer to a transformation matrix for the transfer operation. If there is no matrix for the operation, this parameter is set to `nil`.
`clipRgn`	Contains a handle to the clipping region for the destination image. This region is defined in the destination coordinate system. Apply the clipping region after applying the transformation matrix. If there is no clipping region, this parameter is set to `nil`.
`refCon`	Contains a reference constant value. You can set this value by calling the `SGSetChannelRefCon` function.

The sequence grabber component calls this function only when you are filtering the video data. By filtering the video data through a filter buffer, you can eliminate some visual artifacts that result from noisy input video sources. Your application sets a filter buffer by calling the `SGSetCompressBuffer` function.

If you are using a grab-complete function to determine when frames have been grabbed, you should also implement a grab-compress-complete function (described in the next section). Otherwise, the channel will decompress the specified image before calling your grab-complete function, which will result in significantly lower performance. For details on grab-complete functions, see MyGrabCompleteFunction (page 66).

Error constant	Value	Description
cantDoThatInCurrentMode	-9402	Request invalid in current mode

Your transfer-frame function can use the sequence grabber component's SGTransferFrameForCompress function to support the default behavior.

MyGrabCompressCompleteFunction

The sequence grabber calls your grab-compress-complete function when it is working with a video digitizer that supports compressed source data. Your grab-compress-complete function is responsible for determining whether the current compressed frame has been completely captured and compressed, essentially combining your grab-complete, compress, and compress-complete functions into one function.

Your function must support the following interface:

```
pascal ComponentResult MyGrabCompressCompleteFunction
                                        (SGChannel c,
                                         Boolean *done,
                                         SGCompressInfo *ci,
                                         TimeRecord *tr,
                                         long refCon);
```

Parameter	Description
c	Identifies the channel for this operation.
done	Contains a pointer to a Boolean value. Set this Boolean value to indicate whether you are finished. Set it to true when you are done; set it to false if the operation is incomplete.
ci	Contains a pointer to a compression information structure. When the operation is complete, fill in this structure with information about the compression operation.
tr	Contains a pointer to a time record. When the operation is complete, fill in this structure with information indicating when the frame was grabbed.
refCon	Contains a reference constant value. You can set this value by calling the SGSetChannelRefCon function.

See The Compression Information Structure (page 74), for a description of the SGCompressInfo data type.

Error constant	Value	Description
cantDoThatInCurrentMode	-9402	Request invalid in current mode

Your grab-compress-complete function may use the sequence grabber's SGGrabCompressComplete function to support the default behavior.

MyDisplayCompressFunction

The sequence grabber calls your display-compress function when it is working with a video digitizer component that supports compressed source data. Your display-compress function is responsible for decompressing and displaying a compressed image.

```
pascal ComponentResult MyDisplayCompressFunction (SGChannel c,
                                        Ptr dataPtr,
                                        ImageDescriptionHandle desc,
                                        MatrixRecord *mp,
                                        RgnHandle clipRgn,
                                        long refCon);
```

Parameter	Description
`c`	Identifies the channel for this operation. The sequence grabber provides this value to your display-compress function.
`dataPtr`	Contains a pointer to the compressed image data.
`desc`	Specifies a handle to the image description structure to use for the decompression operation.
`mp`	Contains a pointer to a matrix structure. This matrix structure contains the transformation matrix to use when displaying the image. If there is no matrix for the operation, this parameter is set to `nil`.
`clipRgn`	Contains a handle to the clipping region for the destination image. This region is defined in the destination coordinate system. Apply the clipping region after the transformation matrix. If there is no clipping region, this parameter is set to `nil`.
`refCon`	Contains a reference constant value. You can set this value by calling the `SGSetChannelRefCon` function.

Error constant	Value	Description
`cantDoThatInCurrentMode`	-9402	Request invalid in current mode

Your display-compress function may use the sequence grabber's `SGDisplayCompress` function to support the default behavior.

MyDataFunction

The sequence grabber calls your data function whenever any of the grabber's channels write digitized data to the destination movie file. You assign a data function to the sequence grabber by calling the `SGSetDataProc` function.

Your data function must support the following interface:

```
pascal OSErr MyDataFunction (SGChannel c, Ptr p, long len,
                                        long *offset, long chRefCon,
                                        TimeValue time, short writeType,
```

```
                                    long refCon);
```

Parameter	Description
c	Identifies the channel component that is writing the digitized data.
p	Contains a pointer to the digitized data.
len	Indicates the number of bytes of digitized data.
offset	Contains a pointer to a field that may specify where you are to write the digitized data, and that is to receive a value indicating where you wrote the data. You must update the field referred to by this parameter, supplying the value indicated by the writeType parameter.
chRefCon	Contains control information. The low-order 16 bits contain sample flags for use by the Movie Toolbox's AddMediaSample function. The sequence grabber sets these flags as appropriate. The high-order 16 bits are reserved for Apple and are always set to 0.
time	Identifies the starting time of the data, in the channel's time scale. You may use the SGGetChannelTimeScale function to retrieve the channel's time scale.
writeType	Indicates the type of write operation being performed.
refCon	Contains the reference constant you specified when you assigned your data function to the sequence grabber.

The following values are defined for the writeType parameter:

Constant	Description
seqGrabWriteAppend	Append the new data to the end of the file. Set the field referred to by the offset parameter to reflect the location at which you added the data.
seqGrabWriteReserve	Do not write any data to the output file. Instead, reserve space in the output file for the amount of data indicated by the len parameter. Set the field referred to by the offset parameter to the location of the reserved space.
seqGrabWriteFill	Write the data into the location specified by the field referred to by the offset parameter. Set that field to the location of the byte following the last byte you wrote. This option is used to fill the space reserved previously when the writeType parameter was set to seqGrabWriteReserve. Note that the sequence grabber may call your data function several times to fill a single reserved location.

The sequence grabber calls your data function whenever any channel component writes data to the destination movie. You may use your data function to store the digitized data in some format other than a QuickTime movie.

You can instruct the sequence grabber not to write its data to a QuickTime movie by calling the SGSetDataOutput function and setting the seqGrabDontMakeMovie flag to 1. This can save processing time in cases where you do not want to create or update a movie.

MyModalFilter

The `SGSettingsDialog` function causes the sequence grabber to present its settings dialog box to the user. This is a movable modal dialog box, so you must provide a filter function to handle update events in your window. You specify your filter function with the `proc` parameter.

A modal-dialog filter function whose address is passed to `SGSettingsDialog` should support the following interface:

```
pascal Boolean MyModalFilter (DialogPtr theDialog,
                              EventRecord *theEvent,
                              short *itemHit, long refCon);
```

Parameter	Description
`theDialog`	Points to the settings dialog box's dialog structure.
`theEvent`	Contains a pointer to an event structure. This event structure contains information identifying the nature of the event.
`itemHit`	Contains a pointer to a field that contains the item selected by the user. If you handle the event, you should update this field to reflect the item number of the selected item.
`refCon`	Contains a reference constant. You provide this reference constant to the sequence grabber in the `procRefNum` parameter of the `SGSettingsDialog` function.

Your modal-dialog filter function returns a Boolean value that indicates whether you handled the event. Set this value to `true` if you handled the event; otherwise, set it to `false`. If you handle the event, be sure to update the value of the field referred to by the `itemHit` parameter.

Data Types

This section describes the compression information structure and the sequence grabber frame information structure.

The Compression Information Structure

The compression information structure defines the characteristics of a buffer that contains a captured image that has been compressed. Callback functions use compression information structures to exchange information about compressed images. For example, the compress-complete function must format a compression information record whenever a video frame is compressed (see MyCompressCompleteFunction (page 68) for more information about the compress-complete callback function). The `SGCompressInfo` data type defines a compression information structure.

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`, adding a new `frameOutput` field and extending the `frameOffset` field to 64 bits.

Note: You only need to know about the frame information structure if you are creating a sequence grabber component. If you are not creating a sequence grabber component, you may skip this section.

Sequence Grabber Panel Components

This chapter describes sequence grabber panel components and explains how they are used. Because applications never call sequence grabber panel components directly, only developers planning to create panel components need to read this chapter.

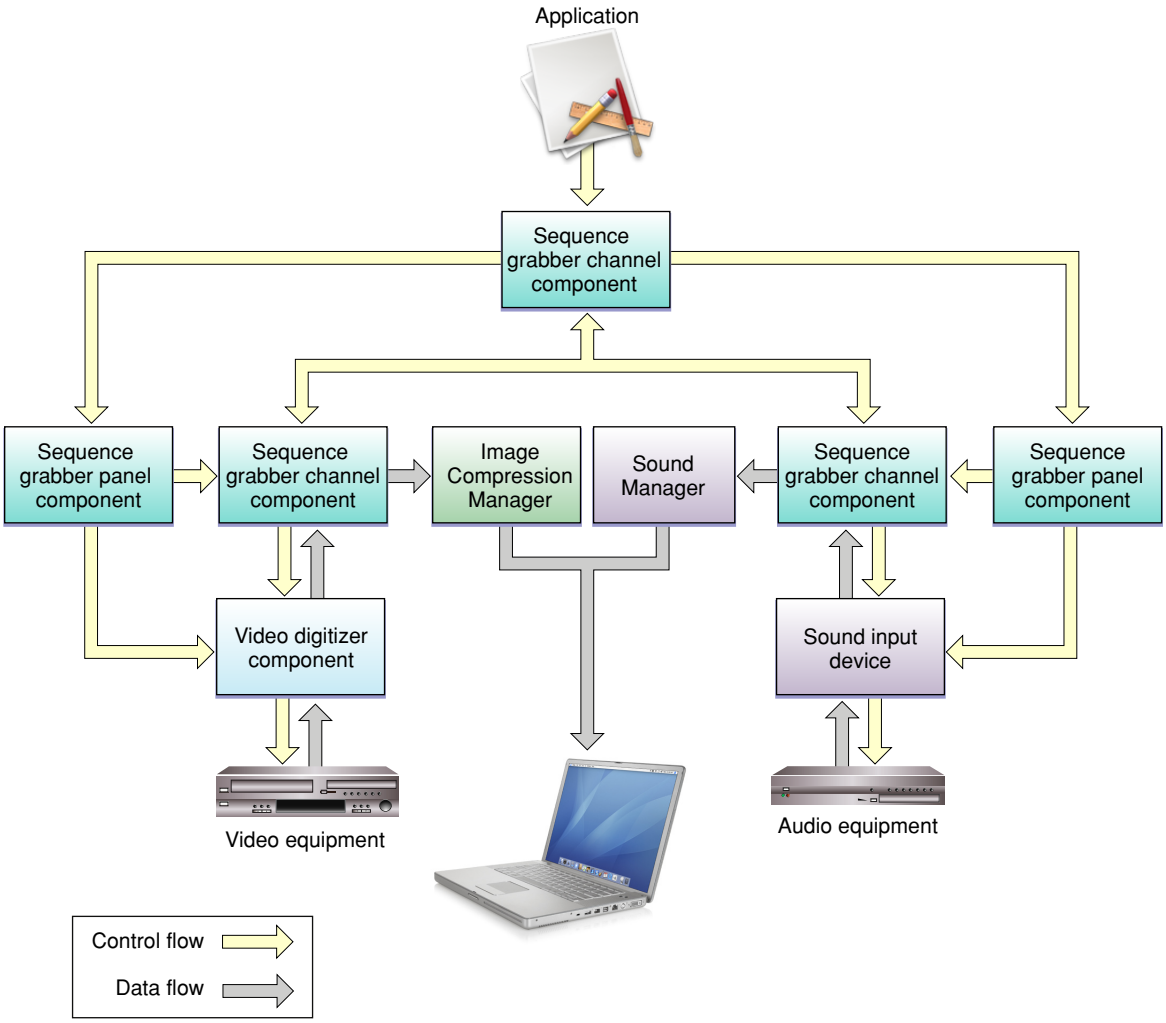
A sequence grabber panel component presents a settings dialog box to the user that affects the behavior of a channel component. For example, a dialog might let the user control the frame capture rate of a video digitizer and the image quality of an image compressor. The user settings dialog box can be customized to include any required options by creating a new component.

How Sequence Grabber Panel Components Work

This section provides background information about sequence grabber panel components. After reading this section, you should understand why these components exist and whether you need to create one.

Sequence grabber panel components augment the capabilities of sequence grabber components and sequence grabber channel components by allowing sequence grabbers to obtain configuration information from the user for a particular digitizing source that is managed by a channel component. Consequently, sequence grabbers, channel components, and panel components have a close relationship.

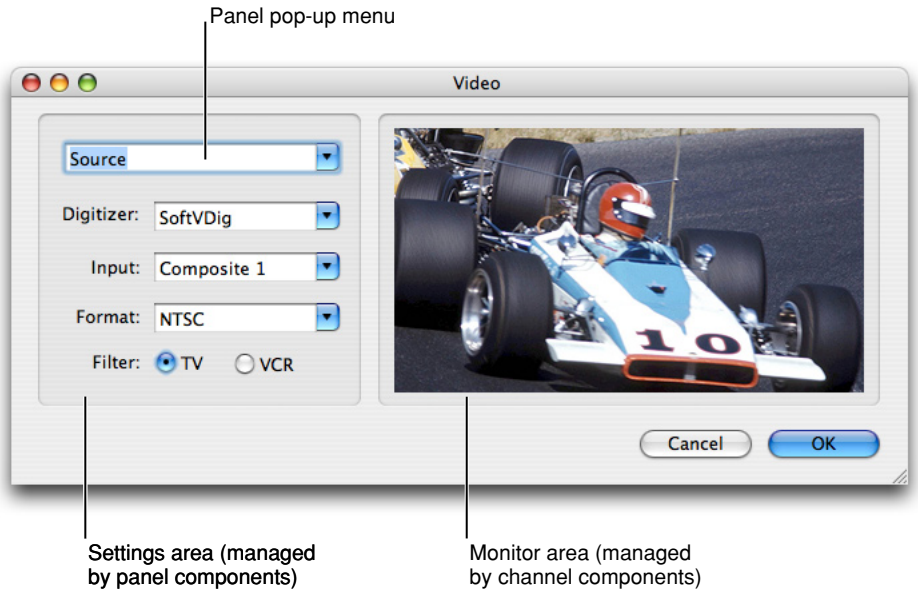
Figure 4-1 shows this relationship and how these components interact with one another to place digitized data into a QuickTime movie.



Sequence grabbers present a settings dialog box to the user whenever an application calls the `SGSettingsDialog` function (see Sequence Grabber Component Functions (page 49) for more information about this sequence grabber function). Applications never call sequence grabber panel components directly; application developers use panel components only by calling the sequence grabber component.

Although the sequence grabber creates the dialog box and manages its interactions with the user, portions of the dialog box are controlled by panel components and channel components.

Figure 4-2 shows a sample dialog box and identifies the various parts of the dialog box.



The sequence grabber creates the dialog box itself and manages the OK and Cancel buttons and the panel pop-up menu. Channel components are responsible for the monitor area on the right side of the dialog box. Panel components manage the settings area immediately below the panel pop-up menu. Only one panel component is active at any given time; the user selects a panel component by manipulating the panel pop-up menu.

When the user selects a specific panel component, the sequence grabber works with that component to build the panel settings dialog area and present it to the user. The panel component processes dialog events and mouse clicks as appropriate and validates the user's settings. The sequence grabber then retrieves the settings from the panel component and stores those settings.

There are two circumstances under which you should consider creating a sequence grabber panel component:

- First, if you want to support special digitizing equipment in the QuickTime environment;
- Second, if you have created your own sequence grabber channel component.

If you have created special digitizing equipment, you may not have to create a special channel component for your equipment; the channel components provided by Apple may be sufficient for your needs. By providing a special panel component, however, you can allow the user to take advantage of your equipment's special capabilities.

If you have created your own channel component, you must create an accompanying panel component to allow the user to configure your channel.

Creating Sequence Grabber Panel Components

This section discusses how to create a sequence grabber panel component. You should read this section if you are creating a panel component.

Applications do not call panel components directly. Rather, they invoke a sequence grabber's settings dialog box by calling the `SGSettingsDialog` function. In response, the sequence grabber presents the settings dialog box to the user. When the user selects a specific settings panel, the sequence grabber invokes the appropriate panel component.

Panel components provide a number of functions that allow sequence grabbers to manage their relationships with panel components. See Managing Your Panel Component (page 81) for complete descriptions of these functions.

Panel components are not responsible for saving their settings information. Sequence grabbers manage this information on behalf of panel components, and a sequence grabber may combine configuration information from several panel components in order to build up the complete configuration for an elaborate digitizing environment. Panel components provide functions that allow sequence grabbers to obtain this configuration information. See Managing Your Panel's Settings (page 81) for more information about these functions.

Sequence grabbers store this configuration data in user data items. The Movie Toolbox provides a number of functions that allow you to create and manage user data items.

Apple has defined a component type value for sequence grabber panel components. You can use the following constant to specify this component type:

```
#define     SeqGrabPanelType 'sgpn'         /* panel component type */
```

Sequence grabber panel components use their component subtype and manufacturer values to indicate the type of configuration services they provide. The subtype value indicates the media type supported by the panel component. This value should correspond to the component subtype value of channel components that may be configured by the panel component. For example, a panel component that manages video settings would have a subtype of `'vide'` (this value is defined by the Movie Toolbox's `VideoMediaType` constant).

The manufacturer field contains a unique identifier for each panel component. The value should indicate something about the specific services provided by the component. For example, Apple has defined the following manufacturer values:

```
#define   SeqGrabCompressionPanelType  'sour'  /* input source selection */
#define   SeqGrabSourcePanelType       'cmpr'  /* compression settings */
```

In general, Apple has reserved all lowercase values of component subtypes and manufacturer codes.

Apple has defined a functional interface for sequence grabber panel components. You may use the following constants to refer to the request codes for each of the functions that your component must support:

```
enum {
    /* sequence grabber panel request codes */
    kSGCPanelGetDitlSelect          = 0x200,    /* SGPanelGetDITL */
    kSGCPanelCanRunSelect           = 0x202,    /* SGPanelCanRun */
    kSGCPanelInstallSelect          = 0x203,    /* SGPanelInstall */
    kSGCPanelEventSelect            = 0x204,    /* SGPanelEvent */
    kSGCPanelItemSelect             = 0x205,    /* SGPanelItem */
    kSGCPanelRemoveSelect           = 0x206,    /* SGPanelRemove */
    kSGCPanelSetGrabberSelect       = 0x207,    /* SGPanelSetGrabber */
    kSGCPanelSetResFileSelect       = 0x208,    /* SGPanelSetResFile */
    kSGCPanelGetSettingsSelect      = 0x209,    /* SGPanelGetSettings */
    kSGCPanelSetSettingsSelect      = 0x20A,    /* SGPanelSetSettings */
    kSGCPanelValidateInputSelect    = 0x20B
};
```

Before reading the rest of this chapter, you should have a basic understanding of how to create components. To create a sequence grabber panel component, you set up the global variables and implement the required Component Manager request codes and the functions that are private to your particular component. Then you manage the dialog box and work with the settings in the dialog box.

Managing Your Panel Component

Sequence grabber components load, configure, and unload your panel component. As part of this process, the sequence grabber installs your panel's dialog items into the settings dialog box and may open your component's resource file. Panel components provide a number of functions that allow the sequence grabber to manage its relationship with panel components. This section discusses those functions.

After opening a connection to your panel component, the sequence grabber identifies itself to your component by calling your `SGPanelSetGrabber` function. The sequence grabber then tries to determine whether your component can work with its associated channel component by calling your `SGPanelCanRun` function. The sequence grabber calls this function only if you have set the `channelFlagHasDependency` component flag to 1.

Once the sequence grabber has determined that your panel component can work with its channel component, the sequence grabber may open your component's resource file (unless you have set the `channelFlagDontOpenResFile` component flag to 1). Once it has opened the resource file, it passes the file's reference number to you by calling your `SGPanelSetResFile` function.

Next, the sequence grabber prepares to add your component's items to the settings dialog box. The sequence grabber obtains your item list by calling your `SGPanelGetDITL` function. Once it has installed the items, it calls your `SGPanelInstall` function, giving you an opportunity to set initial values.

Before the sequence grabber removes your items from the settings dialog box, it calls your `SGPanelRemove` function.

Managing Your Panel's Settings

Sequence grabber components store their configuration information in Movie Toolbox user data items. This configuration information includes settings for each of the channels used by the sequence grabber. Because your panel component configures sequence grabber channels, your panel component is responsible for creating and formatting the contents of its user data items. The sequence grabber component calls your component whenever it wants to retrieve these settings. The sequence grabber may also use previously stored settings to restore your panel's settings. This section discusses the functions that allow the sequence grabber to work with your panel's settings.

The sequence grabber calls your `SGPanelGetSettings` function in order to retrieve your panel's current settings. The sequence grabber uses your `SGPanelSetSettings` function to restore those settings to some previous values.

Component Flags for Sequence Grabber Panel Components

The Component Manager allows you to specify information about your component's capabilities in the `componentFlags` field of the component description record. Sequence grabber panel components use the `componentFlags` field to indicate specific information about their capabilities.

The following flags are defined:

```
enum {
    channelFlagDontOpenResFile = 2,   /* do not open resource file */
    channelFlagHasDependency = 4      /* needs special hardware */
};
```

These flags control how sequence grabbers manage their connection with your panel component. The `channelFlagDontOpenResFile` flag instructs the sequence grabber not to open your component's resource file. By default, the sequence grabber opens your component's resource file for you, and then provides you with the appropriate file reference number. In general, this is convenient. However, if your component is linked with your application and does not have its own resource file, you may not want the sequence grabber to try to open the resource file. In such cases, set this flag to 1.

The `channelFlagHasDependency` flag allows you to tell the sequence grabber that your panel component requires special digitizing hardware. If you set this flag to 1, the sequence grabber gives your component an opportunity to verify that it can work in the current hardware environment by calling your component's `SGPanelCanRun` function.

Processing Your Panel's Events

When your panel component is loaded into the settings dialog box and active, you may receive and process dialog events and mouse clicks.

Your component's `SGPanelEvent` function acts like a modal-dialog filter function, allowing you to process individual dialog events. The sequence grabber calls your `SGPanelItem` function whenever the user clicks a dialog item.

Whenever the user clicks the OK button, the sequence grabber calls your `SGPanelValidateInput` function. Your panel component may then validate the user's settings.

Implementing the Required Component Functions

Listing 4-1 illustrates the component dispatchers for a sequence grabber panel component together with the required functions for open, close, can do, and version.

Listing 4-1 Implementing functions for open, close, can do, and version

```
#define sgcPictShowTicksType 'TICK'

typedef struct {
    ComponentInstance           self;
    ControlHandle               ch;
} PictPanelGlobalsRecord, *PictPanelGlobals;
```

```
/* only for PICT channels */
pascal ComponentResult SGSetShowTickCount (SGChannel c,
                    Boolean show) = {0x2f3c,2,0x100,0x7000,0xA82A};
pascal ComponentResult SGGetShowTickCount (SGChannel c,
                    Boolean *show) = {0x2f3c,4,0x101,0x7000,0xA82A};
pascal ComponentResult PictPanelDispatcher
                    (ComponentParameters *params, Handle storage)

{
    OSErr err = badComponentSelector;
    ComponentFunction componentProc = 0;
    switch (params->what) {
        case kComponentOpenSelect:
            componentProc = PictPanelOpen; break;
        case kComponentCloseSelect:
            componentProc = PictPanelClose; break;
        case kComponentCanDoSelect:
            componentProc = PictPanelCanDo; break;
        case kComponentVersionSelect:
            componentProc = PictPanelVersion; break;
        case kSGCPanelGetDitlSelect:
            componentProc = PictPanelPanelGetDitl; break;
        case kSGCPanelInstallSelect:
            componentProc = PictPanelPanelInstall; break;
        case kSGCPanelItemSelect:
            componentProc = PictPanelPanelItem; break;
        case kSGCPanelRemoveSelect:
            componentProc = PictPanelPanelRemove; break;
        case kSGCPanelGetSettingsSelect:
            componentProc = PictPanelPanelGetSettings; break;
        case kSGCPanelSetSettingsSelect:
            componentProc = PictPanelPanelSetSettings; break;
    }
    if (componentProc)
        err = CallComponentFunctionWithStorage (storage, params,
                                                componentProc);
    return err;
}
pascal ComponentResult PictPanelCanDo (PictPanelGlobals store,
                                       short ftnNumber)
{
    switch (ftnNumber) {
        case kComponentOpenSelect:
        case kComponentCloseSelect:
        case kComponentCanDoSelect:
        case kComponentVersionSelect:
        case kSGCPanelGetDitlSelect:
        case kSGCPanelInstallSelect:
        case kSGCPanelItemSelect:
        case kSGCPanelRemoveSelect:
        case kSGCPanelGetSettingsSelect:
        case kSGCPanelSetSettingsSelect:
            return true;
        default:
            return false;
    }
}
pascal ComponentResult PictPanelVersion (PictPanelGlobals store)
```

```
{
    return 0x00020001;
}

pascal ComponentResult PictPanelOpen (PictPanelGlobals store,
                                      ComponentInstance self)
{
    OSErr err;

    /* allocate global variables */
    store = (PictPanelGlobals) NewPtrClear
                (sizeof(PictPanelGlobalsRecord));
    if (err = MemError()) goto bail;
    SetComponentInstanceStorage (self, (Handle)store);

    /* remember the component instance identification number */
    store->self = self;
bail:
    return err;
}
pascal ComponentResult PictPanelClose (PictPanelGlobals store,
                                       ComponentInstance self)
{
    if (store) DisposePtr ((Ptr)store);
    return noErr;
}
```

Managing the Dialog Box

This section gives details on the functions that a panel component must provide so that the sequence grabber can load the component's items into the settings dialog box and receive and process dialog events.

- To prepare to add the component's items to the settings dialog box, the sequence grabber obtains the item list by calling the `SGPanelGetDITL` function.
- Once it has installed the items, the sequence grabber calls the `SGPanelInstall` function, which sets up the state of the dialog box (for example, a checkbox) and gives the panel component an opportunity to set initial values.
- When the panel component is loaded into the settings dialog box and active, it may receive and process dialog events and mouse clicks. The component's `SGPanelEvent` function processes individual dialog events.
- Whenever the user clicks a dialog item, the sequence grabber calls the `SGPanelItem` function.
- Before the sequence grabber removes the items from the settings dialog box, it calls the `SGPanelRemove` function.

Listing 4-2 provides an example of the management of the settings dialog box for a sequence grabber that displays PICT images. The component item displayed in the dialog box in this case is a tick count checkbox.

Listing 4-2 Managing the settings dialog box

```
pascal ComponentResult PictPanelPanelGetDitl
                        (PictPanelGlobals store, Handle *ditl)
```

```
{
    /*
        Get and detach the dialog box template. Note that
        the sequence grabber has already opened the resource file.
    */
    *ditl = GetResource ('DITL', 7001);
    if (!*ditl) return resNotFound;
    DetachResource (*ditl);
    return noErr;
}

pascal ComponentResult PictPanelPanelInstall
                                (PictPanelGlobals store, SGChannel c,
                                 DialogPtr d, short itemOffset)
{
    Rect r;
    short kind;
    Handle h;
    Boolean ticksShowing;
    /* set up the initial state of the checkbox */
    GetDItem (d, 1 + itemOffset, &kind, &h, &r);
    store->ch = (ControlHandle)h;
    SGGetShowTickCount (c, &ticksShowing);
    SetCtlValue (store->ch, ticksShowing);
    return noErr;
}
pascal ComponentResult PictPanelPanelItem
                                (PictPanelGlobals store, SGChannel c,
                                 DialogPtr d, short itemOffset,
                                 short itemNum)
{
    /* if the item clicked was your checkbox, update its state */
    if ((itemNum - itemOffset) == 1) {
        Boolean showing = GetCtlValue (store->ch);
        SetCtlValue (store->ch, !showing);
        SGSetShowTickCount (c, !showing);
    }
    return noErr;
}
pascal ComponentResult PictPanelPanelRemove
                                        (PictPanelGlobals store,
                                        SGChannel c, DialogPtr d,
                                        short itemOffset)
{
    /* forget that it ever had a control */
    store->ch = nil;
    return noErr;
}
```

To allow the sequence grabber to work with your panel's settings, your panel component must allow the sequence grabber to

- retrieve the panel's current settings by calling your `SGPanelGetSettings` function
- restore those settings to some previous values by using your `SGPanelSetSettings` function

Listing 4-3 gives an example in which the settings are managed in a user list that contains tick count information for a panel component for PICT images.

Listing 4-3 Managing the settings for a panel component

```
pascal ComponentResult PictPanelPanelGetSettings
                           (PictPanelGlobals store, SGChannel c,
                            UserData *result, long flags)
{
    OSErr           err;
    UserData        ud;
    Boolean         ticksShowing;

    /* create a user data list containing your state */
    if (err = NewUserData (&ud)) goto bail;
    if (err = SGGetShowTickCount (c, &ticksShowing)) goto bail;
    if (err = SetUserDataItem (ud, &ticksShowing,
                                sizeof (ticksShowing),
                                sgcPictShowTicksType, 1)) goto bail;
bail:
    if (err) {
        DisposeUserData(ud);
        ud = 0;
    }
    *result = ud;
    return err;
}
pascal ComponentResult PictPanelPanelSetSettings
                          (PictPanelGlobals store, SGChannel c,
                           UserData ud, long flags)
{
    Boolean ticksShowing;
    /* restore the state from the specified user data list */
    if (GetUserDataItem (ud, &ticksShowing,
                                sizeof (ticksShowing),
                                sgcPictShowTicksType, 1) == noErr)
        SGSetShowTickCount (c, ticksShowing);
    return noErr;
}
```

Sequence Grabber Channel Components

This chapter describes how to build sequence grabber channel components, also known simply as **channel components.** These components are used by higher-level sequence grabber components, and act to isolate the sequence grabber from the details of working with actual data types. Channel components may, in turn, depend on the services of still lower-level components, such as video digitizer components.

For example, a sequence grabber component may provide both audio and video to an application. It may receive the audio and video data from two channel components: an audio channel component and a video channel component. The video channel component may receive its data from a video digitizer component that is specific to a particular manufacturer's video capture board.

You should read this chapter if you are developing a sequence grabber component, a channel component, and/or a video digitizer component. Application programmers should use the services of a sequence grabber component, and should not need to read this material.

Creating Sequence Grabber Channel Components

Sequence grabber channel components are the most convenient mechanism for extending the ability of the sequence grabber component to accommodate new types of source data. For example, if you are developing special-purpose hardware that generates a new kind of data, you should create a channel component for that kind of data.

This section discusses issues you should consider when creating a sequence grabber channel component. It also provides a sample program for the implementation of a sequence grabber channel component.

Component Type and Subtype Values

Apple has defined a component type value for sequence grabber channel components; that type value is `'sgch'`. You can use the following constant to specify this type value:

```
#define SeqGrabChannelType 'sgch';
```

Sequence grabber channel components use their component subtype value to indicate the media type created by the component. For example, a channel component that works with video data would have a subtype of `'vide'` (this value is defined by the Movie Toolbox's `VideoMediaType` constant).

Required Functions

At a minimum, your channel component should support the following functions:

- `SGGetChannelInfo`

- SGRelease
- SGGetChannelUsage
- SGSetChannelRefCon
- SGGetDataRate
- SGSetChannelUsage
- SGIdle
- SGStartPreview
- SGInitChannel
- SGStartRecord
- SGPause
- SGStop
- SGPrepare
- SGWriteSamples

In addition, if your channel component supports visual data, it should support at least the following functions:

- SGGetChannelBounds
- SGSetChannelBounds
- SGSetGWorld

If your channel component supports audio data, it should support the following functions as well:

- SGGetChannelVolume
- SGSetChannelVolume

Other functions mentioned in this chapter are optional. However, your channel component should support as many of these functions as possible, so that your component is more useful to applications and users.

Component Manager Request Codes

As with all components, your channel component receives its requests from the Component Manager in the form of request codes. Apple strongly recommends that you fully support all of the Component Manager's request codes in your channel component, especially the target request. Developers will want to extend the capabilities of the sequence grabber channel components. The Component Manager's `CaptureComponent` function, which uses the target request, is the most convenient mechanism for obtaining the services of a component and then extending those services. If your channel component does not support the target request, then it cannot be used by applications or other components in this manner. You can use the following constants to refer to the request codes for each of the functions that your channel component must support.

```
/* basic sequence grabber channel component selectors */
kSGSetGWorldSelect                      = 0x4;
kSGStartPreviewSelect                   = 0x10;
kSGStartRecordSelect                    = 0x11;
kSGIdleSelect                           = 0x12;
```

```
kSGStopSelect                           = 0x13;
kSGPauseSelect                          = 0x14;
kSGPrepareSelect                        = 0x15;
kSGReleaseSelect                        = 0x16;
kSGUpdateSelect                         = 0x27;

/* selectors for common channel configuration functions */
kSGCSetChannelUsageSelect               = 0x80;
kSGCGetChannelUsageSelect               = 0x81;
kSGCSetChannelBoundsSelect              = 0x82;
kSGCGetChannelBoundsSelect              = 0x83;
kSGCSetChannelVolumeSelect              = 0x84;
kSGCGetChannelVolumeSelect              = 0x85;
kSGCGetChannelInfoSelect                = 0x86;
kSGCSetChannelPlayFlagsSelect           = 0x87;
kSGCGetChannelPlayFlagsSelect           = 0x88;
kSGCSetChannelMaxFramesSelect           = 0x89;
kSGCGetChannelMaxFramesSelect           = 0x8a;
kSGCSetChannelRefConSelect              = 0x8b;
kSGCSetChannelClipSelect                = 0x8C;
kSGCGetChannelClipSelect                = 0x8D;
kSGCGetChannelSampleDescriptionSelect   = 0x8E;
kSGCGetChannelDeviceListSelect          = 0x8F;
kSGCSetChannelDeviceSelect              = 0x90;
kSGCSetChannelMatrixSelect              = 0x91;
kSGCGetChannelMatrixSelect              = 0x92;
kSGCGetChannelTimeScaleSelect           = 0x93;

/* selectors for video channel configuration functions */
kSGCGetSrcVideoBoundsSelect             = 0x100;
kSGCSetVideoRectSelect                  = 0x101;
kSGCGetVideoRectSelect                  = 0x102;
kSGCGetVideoCompressorTypeSelect        = 0x103;
kSGCSetVideoCompressorTypeSelect        = 0x104;
kSGCSetVideoCompressorSelect            = 0x105;
kSGCGetVideoCompressorSelect            = 0x106;
kSGCGetVideoDigitizerComponentSelect    = 0x107;
kSGCSetVideoDigitizerComponentSelect    = 0x108;
kSGCVideoDigitizerChangedSelect         = 0x109;
kSGCSetVideoBottlenecksSelect           = 0x10a;
kSGCGetVideoBottlenecksSelect           = 0x10b;
kSGCGrabFrameSelect                     = 0x10c;
kSGCGrabFrameCompleteSelect             = 0x10d;
kSGCDisplayFrameSelect                  = 0x10e;
kSGCCompressFrameSelect                 = 0x10f;
kSGCCompressFrameCompleteSelect         = 0x110;
kSGCAddFrameSelect                      = 0x111;
kSGCTransferFrameForCompressSelect      = 0x112;
kSGCSetCompressBufferSelect             = 0x113;
kSGCGetCompressBufferSelect             = 0x114;
kSGCGetBufferInfoSelect                 = 0x115;
kSGCSetUseScreenBufferSelect            = 0x116;
kSGCGetUseScreenBufferSelect            = 0x117;
kSGCGrabCompressCompleteSelect          = 0x118;
kSGCDisplayCompressSelect               = 0x119;
kSGCSetFrameRateSelect                  = 0x11A;
kSGCGetFrameRateSelect                  = 0x11B;
```

```
    /* selectors for sound channel configuration functions */
    kSGCSetSoundInputDriverSelect           = 0x100;
    kSGCGetSoundInputDriverSelect           = 0x101;
    kSGCSoundInputDriverChangedSelect       = 0x102;
    kSGCSetSoundRecordChunkSizeSelect       = 0x103;
    kSGCGetSoundRecordChunkSizeSelect       = 0x104;
    kSGCSetSoundInputRateSelect             = 0x105;
    kSGCGetSoundInputRateSelect             = 0x106;
    kSGCSetSoundInputParametersSelect       = 0x107;
    kSGCGetSoundInputParametersSelect       = 0x108;

    /* selectors for channel control functions */
    kSGCInitChannelSelect                   = 0x180;
    kSGCWriteSamplesSelect                  = 0x181;
    kSGCGetDataRateSelect                   = 0x182;
    kSGCAlignChannelRectSelect              = 0x183;
};
```

A Sample Sequence Grabber Channel Component

This section describes a sample sequence grabber channel component for PICT image data.

Implementing the Required Component Functions

Listing 5-1 supplies the component dispatchers for the sequence grabber channel component together with the required functions.

Listing 5-1 Setting up global variables and implementing required functions

```
#define kMediaTimeScale 600

typedef struct {
    ComponentInstance       self;
    SeqGrabComponent        grabber;
    long                    usage;
    Boolean                 paused;
    CGrafPtr                destPort;
    GDHandle                destGD;
    CGrafPort               tempPort;
    MatrixRecord            displayMatrix;
    Rect                    destRect;
    Rect                    srcRect;
    RgnHandle               clip;
    Boolean                 inPreview;
    Boolean                 inRecord;
    TimeBase                base;
    long                    bytesWritten;
    Boolean                 showTickCount;
    long                    saveUsage;
} SGPictGlobalsRecord, *SGPictGlobals;

pascal ComponentResult SGPICTDispatcher
                        (ComponentParameters *params, Handle storage)
```

```
{
    OSErr err = badComponentSelector;
    ComponentFunction componentProc = 0;

    switch (params->what) {
        case kComponentOpenSelect:
                componentProc = SGPictOpen; break;
        case kComponentCloseSelect:
                componentProc = SGPictClose; break;
        case kComponentCanDoSelect:
                componentProc = SGPictCanDo; break;
        case kComponentVersionSelect:
                componentProc = SGPictVersion; break;
        case kSGSetGWorldSelect:
                componentProc = SGPictSetGWorld; break;
        case kSGStartPreviewSelect:
                componentProc = SGPictStartPreview; break;
        case kSGStartRecordSelect:
                componentProc = SGPictStartRecord; break;
        case kSGIdleSelect:
                componentProc = SGPictIdle; break;
        case kSGStopSelect:
                componentProc = SGPictStop; break;
        case kSGPauseSelect:
                componentProc = SGPictPause; break;
        case kSGPrepareSelect:
                componentProc = SGPictPrepare; break;
        case kSGReleaseSelect:
                componentProc = SGPictRelease; break;
        case kSGCSetChannelUsageSelect:
                componentProc = SGPictSetChannelUsage; break;
        case kSGCGetChannelUsageSelect:
                componentProc = SGPictGetChannelUsage; break;
        case kSGCSetChannelBoundsSelect:
                componentProc = SGPictSetChannelBounds; break;
        case kSGCGetChannelBoundsSelect:
                componentProc = SGPictGetChannelBounds; break;
        case kSGCGetChannelInfoSelect:
                componentProc = SGPictGetChannelInfo; break;
        case kSGCSetChannelMatrixSelect:
                componentProc = SGPictSetChannelMatrix; break;
        case kSGCGetChannelMatrixSelect:
                componentProc = SGPictGetChannelMatrix; break;
        case kSGCSetChannelClipSelect:
                componentProc = SGPictSetChannelClip; break;
        case kSGCGetChannelClipSelect:
                componentProc = SGPictGetChannelClip; break;
        case kSGCGetChannelSampleDescriptionSelect:
                componentProc = SGPictGetChannelSampleDescription;
                 break;
        case kSGCGetChannelDeviceListSelect:
            componentProc = SGPictGetChannelDeviceList; break;
        case kSGCSetChannelDeviceSelect:
            componentProc = SGPictSetChannelDevice; break;
        case kSGCGetChannelTimeScaleSelect:
            componentProc = SGPictGetChannelTimeScale; break;
        case kSGCInitChannelSelect:
            componentProc = SGPictInitChannel; break;
```

```
        case kSGCWriteSamplesSelect:
            componentProc = SGPictWriteSamples; break;
        case kSGCGetDataRateSelect:
                componentProc = SGPictGetDataRate; break;
        case kSGCPanelGetDitlSelect:
                componentProc = SGPictPanelGetDitl; break;
        case kSGCPanelInstallSelect:
                componentProc = SGPictPanelInstall; break;
        case kSGCPanelEventSelect:
                componentProc = SGPictPanelEvent; break;
        case kSGCPanelRemoveSelect:
                componentProc = SGPictPanelRemove; break;
        case kSGCPanelGetSettingsSelect:
                componentProc = SGPictPanelGetSettings; break;
        case kSGCPanelSetSettingsSelect:
                componentProc = SGPictPanelSetSettings; break;
        case 0x0100:
                componentProc = SGPictSetShowTickCount; break;
        case 0x0101:
                componentProc = SGPictGetShowTickCount; break;
    }
    if (componentProc)
        err = CallComponentFunctionWithStorage (storage, params,
                                                componentProc);
    return err;
}

pascal ComponentResult SGPictCanDo (SGPictGlobals store,
                                    short ftnNumber)
{
    switch (ftnNumber) {
        case kComponentOpenSelect:
        case kComponentCloseSelect:
        case kComponentCanDoSelect:
        case kComponentVersionSelect:
        case kSGSetGWorldSelect:
        case kSGStartPreviewSelect:
        case kSGStartRecordSelect:
        case kSGIdleSelect:
        case kSGStopSelect:
        case kSGPauseSelect:
        case kSGPrepareSelect:
        case kSGReleaseSelect:
        case kSGCSetChannelUsageSelect:
        case kSGCGetChannelUsageSelect:
        case kSGCSetChannelBoundsSelect:
        case kSGCGetChannelBoundsSelect:
        case kSGCGetChannelInfoSelect:
        case kSGCSetChannelMatrixSelect:
        case kSGCGetChannelMatrixSelect:
        case kSGCSetChannelClipSelect:
        case kSGCGetChannelClipSelect:
        case kSGCGetChannelSampleDescriptionSelect:
        case kSGCGetChannelDeviceListSelect:
        case kSGCSetChannelDeviceSelect:
        case kSGCGetChannelTimeScaleSelect:
        case kSGCInitChannelSelect:
        case kSGCWriteSamplesSelect:
```

```
        case kSGCGetDataRateSelect:
        case kSGCPanelGetDitlSelect:
        case kSGCPanelInstallSelect:
        case kSGCPanelEventSelect:
        case kSGCPanelRemoveSelect:
        case kSGCPanelGetSettingsSelect:
        case kSGCPanelSetSettingsSelect:

        /* private component functions */
        case 0x0100:
        case 0x0101:
            return true;
        default:
            return false;
    }
}

pascal ComponentResult SGPictVersion (SGPictGlobals store)
{
    return 0x00020001;
}

pascal ComponentResult SGPictOpen (SGPictGlobals store,
                                   ComponentInstance self)
{
    OSErr err;
    GrafPtr savePort;

    /* allocate global variables */
    store =
    (SGPictGlobals)NewPtrClear(sizeof(SGPictGlobalsRecord));
    if (err = MemError()) goto bail;

    /* create a temporary port for drawing during the idle
        function */
    GetPort (&savePort);
    OpenCPort (&store->tempPort);
    SetPort ((GrafPtr)&store->tempPort);
    PortSize (4096, 4096);
    SetRectRgn (store->tempPort.visRgn, 0, 0, 4096, 4096);
    ClipRgn (store->tempPort.visRgn);
    SetPort (savePort);

    store->self = self;
    store->showTickCount = false;
    SetComponentInstanceStorage (self, (Handle)store);

bail:
    return err;
}

pascal ComponentResult SGPictClose (SGPictGlobals store,
                                    ComponentInstance self)
{
    /* disposal operations */
    if (store) {
        if (store->clip) DisposeRgn(store->clip);
        CloseCPort(&store->tempPort);
```

```
        DisposPtr((Ptr)store);
    }
    return noErr;
}
```

Initializing the Sequence Grabber Channel Component

To initialize the channel component, the sequence grabber component calls the `SGInitChannel` function.

The code in Listing 5-2 initializes channel variables. The grabber component calls the `SGPictInitChannel` function to initialize a sequence grabber channel component. The `SGPictInitChannel` function calls QuickDraw's `SetRect` routine and QuickTime's `SetIdentityMatrix` function to specify the size of the area (around a mouse-down event) in which the sequence grabber component will capture PICT images.

Listing 5-2 Initializing the sequence grabber channel component

```
pascal ComponentResult SGPictInitChannel (SGPictGlobals store,
                                          SeqGrabComponent owner)
{
    /* initialize any variables here */
    SetRect(&store->srcRect, 0, 0, 160, 120);/* rectangle in which
                                                capture occurs */
    SetIdentityMatrix (&store->displayMatrix);

    store->grabber = owner;
    SGGetTimeBase (owner, &store->base);

    return noErr;
}
```

Setting and Retrieving the Channel State

Listing 5-3 supplies configuration functions that set the usage parameters and storage for the channel component. (See the descriptions of the `SGSetChannelUsage` and `SGGetChannelUsage` functions for details.)

The sample code illustrates how to retrieve usage information. In this case, you indicate that the sequence grabber component has spatial boundaries by using the `seqGrabHasBounds` constant in the `channelInfo` parameter.

Listing 5-3 Determining usage parameters and getting usage data

```
pascal ComponentResult SGPictSetChannelUsage(SGPictGlobals store, long usage)
{
    /* remember usage */
    store->usage = usage;
    return noErr;
}

pascal ComponentResult SGPictGetChannelUsage(SGPictGlobals store, long *usage)
{
    /* return usage */
    *usage = store->usage;
```

```
    return noErr;
}

pascal ComponentResult SGPictGetChannelInfo (SGPictGlobals store,
                                                long *channelInfo)
{
    /* indicate that you have spatial boundaries */
    *channelInfo = seqGrabHasBounds;
    return noErr;
}
```

Managing Spatial Properties

To set up an area in which the channel component displays image data, the sequence grabber should perform these tasks:

1. Assign the destination graphics world and graphics device for the display of the captured image with the `SGSetGWorld` function.
2. Specify a display transformation matrix for a video channel using the `SGSetChannelMatrix` function. Your function determines the matrix that is being set, validates it, and updates the matrix and destination rectangle. Your channel uses this matrix to transform its video image into the destination window.
3. Obtain the channel's display transformation matrix by calling the `SGGetChannelMatrix` function.
4. Specify the channel's display boundary rectangle with the `SGSetChannelBounds` function. The display boundary rectangle defines the destination for data from this channel and adjusts the channel matrix.
5. Determine the channel's display boundary rectangle with the `SGGetChannelBounds` function.
6. Dispose of the old clipping region and apply a new clipping region to the channel's display region using the `SGSetChannelClip` function.
7. Retrieve the new clipping region by calling the `SGGetChannelClip` function.

The code in Listing 5-4 provides an example of how to manage the spatial characteristics of the area in which the channel component displays PICT image data.

Listing 5-4 Managing spatial characteristics

```
pascal ComponentResult SGPictSetGWorld (SGPictGlobals store,
                                        CGrafPtr gp, GDHandle gd)
{
    /* remember the destination graphics world */
    store->destPort = gp;
    store->destGD = gd;
    return noErr;
}

pascal ComponentResult SGPictSetChannelMatrix
                        (SGPictGlobals store, const MatrixRecord *m)
{
    OSErr err = noErr;
    MatrixRecord mat;
```

```
    short matType;

    /* determine the matrix being set */
    if (m)
        mat = *m;
    else
        SetIdentityMatrix (&mat);

    /* validate it */
    matType = GetMatrixType (&mat);

    if ((mat.matrix[0][0] < 0) || (mat.matrix[1][1] < 0) ||
    (matType >= linearMatrixType))
        return paramErr;

    /* update the matrix and destination rectangle */
    store->displayMatrix = mat;
    store->destRect = store->srcRect;
    TransformRect (&mat, &store->destRect, nil);
    return err;
}

pascal ComponentResult SGPictGetChannelMatrix
                        (SGPictGlobals store, MatrixRecord *m)
{
    /* return current matrix */
    *m = store->displayMatrix;
    return noErr;
}

pascal ComponentResult SGPictSetChannelBounds
                          (SGPictGlobals store, const Rect *bounds)
{
    /* remember destination rect */
    store->destRect = *bounds;

    /* recalculate display matrix from it */
    RectMatrix (&store->displayMatrix, &store->srcRect,
                    &store->destRect);
    return noErr;
}

pascal ComponentResult SGPictGetChannelBounds
                           (SGPictGlobals store, Rect *bounds)
{
    /* return current boundaries */
    *bounds = store->destRect;
    return noErr;
}

pascal ComponentResult SGPictSetChannelClip (SGPictGlobals store,
                                             RgnHandle theClip)
{
    OSErr err = noErr;

    /* toss the old channel clipping */
    if (store->clip) {
        DisposeRgn (store->clip);
```

```
        store->clip = nil;
    }
    /* and remember the new one */
    if (theClip) {
        err = HandToHand ((Handle *)&theClip);
        store->clip = theClip;
    }
    return err;
}

pascal ComponentResult SGPictGetChannelClip
                        (SGPictGlobals store, RgnHandle *theClip)
{
    OSErr err = noErr;

    /* return clip, if there is one */
    if (*theClip = store->clip)
        err = HandToHand ((Handle *)theClip);
    return err;
}
```

Controlling Previewing and Recording Operations

To preview and record image data in the channel component, the code in Listing 5-5 implements these tasks:

1. The `SGStartPreview` function instructs the channel to commence processing any source data. In preview mode, the component does not save any of the data it gathers from its source. Your channel component should immediately present the data to the user in the appropriate format for the channel's configuration and display video data in the destination display region.

2. The `SGStartRecord` function instructs the channel to begin recording data from its source. The sequence grabber component stores the collected data. The channel component should immediately begin recording data.

3. The `SGIdle` function allows the sequence grabber component to grant processing time to the channel component. The `SGIdle` function permits the processing time for the previewing and recording operations to take place. In the example shown in Listing 5-5, the work for the channel consists of getting the current time, adding data to the movie if recording, and showing the preview image if necessary.

4. The `SGStop` function stops the channel's preview and recording operations.

5. The `SGPause` function suspends or restarts the channel's preview and recording operations.

6. The `SGPrepare` function has the sequence grabber component prepare the channel for subsequent preview or record operations.

7. The `SGRelease` function releases any system resources that were allocated during preview or recording operations and that remain thereafter.

The code in Listing 5-5 illustrates a channel component's control of the previewing and recording of a PICT image.

Listing 5-5 Controlling previewing and recording operations

```
pascal ComponentResult SGPictStartPreview (SGPictGlobals store)
{
    /* into preview mode */
    store->inPreview = (store->usage & seqGrabPreview) != 0;
    return noErr;
}

pascal ComponentResult SGPictStartRecord (SGPictGlobals store)
{
    /* into record mode (also preview, if PlayDuringRecord) */
    store->inRecord = (store->usage & seqGrabRecord) != 0;
    store->inPreview = (store->usage & seqGrabPlayDuringRecord) !=
    0;
    return noErr;
}

pascal ComponentResult SGPictIdle (SGPictGlobals store)
{
    OSErr err = noErr;

    /* this is where the work for preview and record happens */
    if (!store->paused && (store->inRecord || store->inPreview)) {
        Point mouseLoc;
        Rect r;
        PicHandle tempPict = nil;
        TimeRecord tr;
        CGrafPtr savePort;
        GDHandle saveGD;
        Rect maxR;

        GetGWorld (&savePort, &saveGD);

        /* get the current time */
        GetTimeBaseTime (store->base, kMediaTimeScale, &tr);

        /* figure the current area around the mouse
            (only on main screen) */
        SetGWorld (&store->tempPort, GetMainDevice());
        GetMouse (&mouseLoc);
        LocalToGlobal (&mouseLoc);
        r.top = r.bottom = mouseLoc.v;
        r.left = r.right = mouseLoc.h;
        InsetRect(&r, -(store->srcRect.right >> 1),
                        -(store->srcRect.bottom >> 1));
        maxR = (**GetMainDevice()).gdRect;
        if (r.left < maxR.left)
             OffsetRect (&r, -r.left + maxR.left, 0);
        if (r.top < maxR.top)
             OffsetRect (&r, 0, -r.top + maxR.top);
        if (r.right > maxR.right)
             OffsetRect(&r, maxR.right - r.right, 0);
        if (r.bottom > maxR.bottom)
             OffsetRect (&r, 0, maxR.bottom - r.bottom);

        /* copy the screen into a picture */
        tempPict = OpenPicture(&r);
```

```
            CopyBits ((BitMap *)&store->tempPort.portPixMap,
                        (BitMap *)&store->tempPort.portPixMap, &r, &r,
                                    srcCopy, nil);
            if (store->showTickCount) {
                /* if users want to see ticks, draw them */
                Str63 str;
                NumToString ( TickCount(), str);
                /* do some magic positioning */
                r.right = r.left + StringWidth(str) + 4;
                r.bottom = r.top + 14;
                EraseRect (&r);
                MoveTo(r.left + 2, r.bottom - 3);
                TextSize (12);
                DrawString (str);
            }
        ClosePicture();

        /* if recording, add data to movie */
        if (store->inRecord) {
            long offset;
            long pictSize = GetHandleSize ((Handle)tempPict);

            HLock ((Handle)tempPict);
            err = SGAddMovieData (store->grabber, store->self,
                                    (Ptr)*tempPict, pictSize, &offset, 0,
                                    tr.value.lo, seqGrabWriteAppend);
            store->bytesWritten += pictSize;
        }

        /* if you need to show the preview image, do that */
        if (store->inPreview) {
            RgnHandle saveClip;
            SetGWorld (store->destPort, store->destGD);
            if (store->clip) {
                saveClip = NewRgn();
                GetClip (saveClip);
                SetClip (store->clip);
            }
            DrawPicture (tempPict, &store->destRect);
            if (store->clip) {
                SetClip (saveClip);
                DisposeRgn (saveClip);
            }
        }

        KillPicture (tempPict);

        SetGWorld (savePort, saveGD);
    }
    return err;
}

pascal ComponentResult SGPictStop (SGPictGlobals store)
{
    /* stop all previewing and recording */
    store->inRecord = store->inPreview = false;
    return noErr;
}
```

```
pascal ComponentResult SGPictPause (SGPictGlobals store,
                                                Byte pause)
{
    /* pause */
    store->paused = pause;
    return noErr;
}

pascal ComponentResult SGPictPrepare (SGPictGlobals store,
                                            Boolean prepareForPreview,
                                            Boolean prepareForRecord)
{
    /* prepare for previewing and recording operations --
        all you do here is initialize a variable */
    store->bytesWritten = 0;
    return noErr;
}

pascal ComponentResult SGPictRelease (SGPictGlobals store)
{
    /* no resources to release after previewing or recording */
    return noErr;
}
```

Managing Channel Devices

To manage channel devices such as video digitizers or sound input drivers, you should

1. Let the sequence grabber retrieve a list of devices that are valid for the channel, using the `SGGetChannelDeviceList` function.
2. Assign an appropriate channel device with the `SGSetChannelDevice` function.

Listing 5-6 provides examples of these required functions for channel device management. The `SGPictGetChannelDeviceList` function obtains a list of devices associated with the channel component. The `SGPictSetChannelDevice` function allows the sequence grabber to specify a channel device. In this code sample, there are no devices associated with the channel component.

Listing 5-6 Coordinating devices for the channel component

```
pascal ComponentResult SGPictGetChannelDeviceList
                                (SGPictGlobals store,
                                 long selectionFlags,
                                 SGDeviceList *list)
{
    *list = (SGDeviceList) NewHandleClear
                        (sizeof (SGDeviceListRecord)); /* no devices */
    return MemError();
}

pascal ComponentResult SGPictSetChannelDevice
                                    (SGPictGlobals store, StringPtr name)
{
    /* you have no devices, so no problem */
```

```
    return noErr;
}
```

Utility Functions for Recording Image Data

To record image data, the channel component must allow the sequence grabber to do the following:

- Obtain an appropriate time scale with the `SGGetChannelTimeScale` function.
- Retrieve the sample description of the image that is to be recorded with the `SGGetChannelSampleDescription` function.
- Create a track and media in which to record the sample image by calling the `SGWriteSamples` function. `SGWriteSamples` writes the captured data to a movie file after a record operation.
- Obtain references from the sequence grabber and add them to the newly created media using the `SGGetNextFrameReference` function so that the channel component can retrieve the sample references it stored.
- Determine how many bytes of captured data the channel is collecting each second using the `SGGetDataRate` function.

The code in Listing 5-7 shows how the channel component uses these utility functions to record PICT image data.

Listing 5-7 Recording image data

```
pascal ComponentResult SGPictGetChannelTimeScale
                        (SGPictGlobals store, TimeScale *scale)
{
    *scale = kMediaTimeScale; /* a reasonable default time scale */
    return noErr;
}

pascal ComponentResult SGPictGetChannelSampleDescription
                        (SGPictGlobals store, Handle sampleDesc)
{
    OSErr err;
    SampleDescriptionPtr sdp;

    SetHandleSize (sampleDesc, sizeof(SampleDescription));
    if (err = MemError()) goto bail;

    /* make up a minimal sample description */
    sdp = (SampleDescriptionPtr)*sampleDesc;
    sdp->descSize = sizeof(SampleDescription);
    sdp->dataFormat = 'PICT';
    sdp->resvd1 = 0;
    sdp->resvd2 = 0;
    sdp->dataRefIndex = 0;

bail:
    return err;
}

pascal ComponentResult SGPictWriteSamples (SGPictGlobals store,
```

```
                                                Movie m, AliasHandle theFile)
{
    OSErr err = 0;
    Track pictT;
    Media pictM;
    long i;
    MatrixRecord aMatrix;
    Rect from, to;
    seqGrabFrameInfo fi;
    TimeRecord tr;
    TimeValue mediaDuration;
    SampleDescriptionHandle sampleDesc = 0;

    /* after SGStop, this function creates the track and media */
    if (!(store->usage & seqGrabRecord))
        return err;

    /* get the sample description */
    sampleDesc = (SampleDescriptionHandle)NewHandle(4);
    if (err = MemError()) goto bail;
    if (err = SGGetChannelSampleDescription (store->self,
                                             (Handle)sampleDesc)) goto bail;

    /* figure out the track matrix */
    SetRect (&from, 0, 0, store->srcRect.right,
             store->srcRect.bottom);
    to = from;

    TransformRect (&store->displayMatrix, &to, nil);

    /* create the track and media */
    pictT = NewMovieTrack (m, (long)from.right << 16,
                               (long)from.bottom << 16, 0);
    pictM = NewTrackMedia (pictT, 'PICT', kMediaTimeScale,
                               (Handle)theFile, rAliasType);

    /* spin in a loop getting sample references from the
        sequence grabber and adding them to the media */
    fi.frameChannel = store->self;
    i = -1;
    do {
        TimeValue frameDuration;

        err = SGGetNextFrameReference (store->grabber,
                                       &fi, &frameDuration, &i);
        if (err) {
            if (err == paramErr)
                err = 0;
            break;
        }

        err = AddMediaSampleReference (pictM,
                fi.frameOffset, fi.frameSize,
                frameDuration,
                sampleDesc, 1,
                0, 0);

        if (err == invalidDuration) {
```

```
            err = noErr;
            break;
        }
    } while (!err);

done:
    if (err) goto bail;

    GetTimeBaseTime (store->base, 0, &tr);
    ConvertTimeScale (&tr, kMediaTimeScale);
    /* trim media inserted to not extend beyond end time */
    mediaDuration = GetMediaDuration(pictM);

    /* add media to track */
    err = InsertMediaIntoTrack (pictT, 0, 0, tr.value.lo, kFix1);

    /* set track matrix */
    RectMatrix (&aMatrix, &from, &to);
    SetTrackMatrix (pictT, &aMatrix);

    /* set track clipping region */
    SetTrackClipRgn (pictT, store->clip);

bail:
    if (sampleDesc) DisposHandle ((Handle)sampleDesc);
    return err;
}

pascal ComponentResult SGPictGetDataRate (SGPictGlobals store,
                                          long *bytesPerSecond)
{
    /* take a guess at the data rate */
    *bytesPerSecond = 24 * 1024;
    if (store->bytesWritten) {
        TimeValue timeNow = GetTimeBaseTime (store->base, 8, nil);
                            /* one-eighth second resolution */

        if (!timeNow)
            return seqGrabInfoNotAvailable;

        *bytesPerSecond = (store->bytesWritten / timeNow) * 8;
                            /* convert back to seconds */
    }
    return noErr;
}
```

Providing Media-Specific Functions

The channel can provide media-specific functions for a particular channel type. These functions are analogous to the `SGSetVideoCompressorType` and `SGGetVideoCompressorType` functions. These functions allow the sequence grabber to specify and determine the type of image compression the channel component is to apply to the captured video images.

The code in Listing 5-8 provides two specialized channel component functions, `SGPictSetShowTickCount` and `SGPictGetShowTickCount`, which set and retrieve the tick count, respectively. Note that both the functions refer to the `showTickCount` field in the `SGPictGlobals` structure.

Listing 5-8 Showing the tick count

```
pascal ComponentResult SGPictSetShowTickCount
                        (SGPictGlobals store, Boolean show)
{
    store->showTickCount = show;
    return noErr;
}

pascal ComponentResult SGPictGetShowTickCount
                        (SGPictGlobals store, Boolean *show)
{
    *show = store->showTickCount;
    return noErr;
}
```

Managing the Settings Dialog Box

The channel allows the sequence grabber to manage the placement of your channel data in the sequence grabber's settings dialog box. This is how it works:

1. To prepare to add the channel component's items to the settings dialog box, the sequence grabber obtains your item list by calling the sequence grabber panel component's `SGPanelGetDITL` function. It retrieves and detaches the dialog box template from the sequence grabber panel component.
2. Once it has installed the items, the sequence grabber uses the `SGPanelInstall` function so initial values can be set. This function resets the channel to use the dialog window and preview mode. It also updates the boundaries to match the size of the user item list.
3. To provide idle time in which to draw the channel's information in the settings dialog box, the sequence grabber uses the `SGPanelEvent` function. It allows the sequence grabber component to receive and process dialog events in a manner similar to a modal-dialog filter function. In this example, the information is the tick count.
4. Prior to the removal of items from the settings dialog box, the sequence grabber component calls the `SGPanelRemove` function. The sequence grabber supplies information that specifies the channel that the panel is to configure, the dialog box, and the offset of the panel's items into the dialog box.

The code in Listing 5-9 calls the sequence grabber panel component and indicates that the channel component will display a tick count checkbox in the panel settings.

Listing 5-9 Including a tick count checkbox in a dialog box in the panel component

```
pascal ComponentResult SGPictPanelGetDitl (SGPictGlobals store,
                                           Handle *ditl)
{
    /* get and detach your dialog template */
    *ditl = GetResource('DITL', 7000);
    if (!*ditl) return resNotFound;
    DetachResource(*ditl);
    return noErr;
}

pascal ComponentResult SGPictPanelInstall (SGPictGlobals store,
```

```
                                                SGChannel c,
                                                DialogPtr d,
                                                short itemOffset)
{
    Rect newBounds;
    short kind;
    Handle h;

    /* reset this channel to use the dialog window and be in
        preview mode with no clip */
    SGSetGWorld (store->self, (CGrafPtr)d, GetMainDevice());
    SGGetChannelUsage (store->self, &store->saveUsage);
    SGSetChannelUsage (store->self, seqGrabPreview);
    SGSetChannelClip (c, nil);

    /* update boundaries to match size of user item */
    GetDItem (d, 1 + itemOffset, &kind, &h, &newBounds);
    SGSetChannelBounds (c, &newBounds);
    SGStartPreview (store->self);
    return noErr;
}

pascal ComponentResult SGPictPanelEvent (SGPictGlobals store,
                                         SGChannel c, DialogPtr d,
                                         short itemOffset,
                                         EventRecord *theEvent,
                                         short *itemHit,
                                         Boolean *handled)
{
    /* use idle time to draw */
    if (theEvent->what == nullEvent)
        return SGIdle (store->self);

    return noErr;
}

pascal ComponentResult SGPictPanelRemove (SGPictGlobals store,
                                         SGChannel c, DialogPtr d,
                                         short itemOffset)
{
    /* stop playing */
    SGStop (store->self);
    SGRelease (store->self);

    /* note that the clip and bounds are automatically restored
     for you because you stored them using the SGGetSettings
     function */

    /* restore usage */
    SGSetChannelUsage(store->self, store->saveUsage);

    return noErr;
}
```

Displaying Channel Information in the Settings Dialog Box

The final step in the implementation of a sequence grabber channel component is the display of the channel preview in the settings dialog box. Two sequence grabber functions, `SGSettingsDialog` and `SGGetSettingsDialog`, facilitate this process.

1. The channel component instructs the sequence grabber to display its settings dialog box to the user by calling the sequence grabber component's `SGSettingsDialog` function. The user can specify the configuration of a sequence grabber channel in this dialog box.
2. To retrieve the current settings of all channels used by the sequence grabber, call the `SGGetSettings` function. The sequence grabber places all of this configuration information into a Movie Toolbox user data list.

Listing 5-10 illustrates code that creates a user data list to contain the tick count information for the sequence grabber's settings dialog box, adds a matrix to the list, and stores clipping information (if any exists). The sample code then restores the clipping and the matrix.

Listing 5-10 Displaying channel settings

```
pascal ComponentResult SGPictPanelGetSettings
                        (SGPictGlobals store, SGChannel c,
                         UserData *result, long flags)
{
    OSErr err = noErr;
    UserData ud = 0;
    MatrixRecord matrix;
    RgnHandle clip;

    /* create a user data list to hold your state */

    if (err = NewUserData (&ud)) goto bail;

    /* add matrix to user data */

    if (SGGetChannelMatrix (c, &matrix) == noErr) {
        if (err = SetUserDataItem (ud, &matrix, sizeof(matrix),
                                   sgMatrixType, 1))
            goto bail;
    }

    /* store clip, if there is one */
    if (SGGetChannelClip (c, &clip) == noErr) {
        if (clip)
            err = AddUserData (ud, (Handle)clip, sgClipType);
        else
            err = SetUserDataItem (ud, nil, 0, sgClipType, 1);
                                /* add a dummy to indicate none */
        DisposeRgn(clip);
        if (err) goto bail;
    }
bail:
    if (err) {
        DisposeUserData (ud);
        ud = 0;
```

```
    }
    *result = ud;
    return err;
}

pascal ComponentResult SGPictPanelSetSettings
                                    (SGPictGlobals store,
                                     SGChannel c, UserData ud, long flags)
{
    OSErr err;
    RgnHandle clip = NewRgn();
    MatrixRecord matrix;

    /* restore clip, if one was stored */
    if (GetUserData (ud, (Handle)clip, sgClipType, 1) == noErr) {
        if (err = SGSetChannelClip
                        (c, GetHandleSize ((Handle)clip) ? clip : 0))
            goto bail;
    }

    /* restore matrix */
    if (err = GetUserDataItem (ud, &matrix, sizeof(matrix),
                                        sgMatrixType, 1)) goto bail;
    if (err = SGSetChannelMatrix (c, &matrix))
         goto bail;

bail:
    DisposeRgn (clip);
    return err;
}
```

Support for Sound Capture at Any Sample Rate

The sequence grabber sound channel allows sound to be captured at any sample rate. The sample rate is specified 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.

Channel Source Names

The sequence grabber supports two functions, `SGChannelSetDataSourceName` and `SGChannelGetDataSourceName`, that allow you to specify the source identification information associated with a sequence grabber channel.

Capturing to Multiple Files

In QuickTime, 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 5-11. 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 5-11 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.

Using Sequence Grabber Channel Components

This chapter gives an overview of the services your channel component needs to provide. Your component will primarily be used to preview and record digital data. Your component must also make calls to application-defined callback functions if so directed. Finally, your component must provide utility functions that will perform default procedures for the application's callback functions.

In response to application requests, sequence grabber components can use channel components in two ways: to preview digitized data for the user or to record captured data in a QuickTime movie. Applications can use previewing to allow the user to prepare to make a recording. Applications that use previewing can move directly from the preview operation to a record operation, without stopping the process.

The next two sections provide an overview of preview and record operations. A third section discusses the callback functions that are supported by some channel components.

Previewing

Previewing captured data involves playing that data for the user as it is digitized. For video data, this means displaying the video images on the computer screen. For audio data, this means playing the sound through the computer's sound system. The following paragraphs outline the steps the sequence grabber component follows to preview captured data.

1. First, the sequence grab ber component opens a connection to your channel component, using the Component Manager's `OpenComponent` function. The sequence grabber component then calls your `SGInitChannel` function to initialize your component.

2. The sequence grabber component then configures your channel component for the preview operation. The `SGSetGWorld` function sets the graphics world in which the preview is to be displayed. The `SGSetChannelUsage` function specifies that your channel is to be used for previewing. The application can then use the appropriate channel configuration functions to prepare your channel for the preview operation. For video channels, it uses the functions discussed in Configuration Functions for Video Channel Components (page 113). For sound channels, the sequence grabber uses the functions discussed in Configuration Functions for Sound Channel Components (page 114).

3. The sequence grabber component starts the preview operation by calling your `SGStartPreview` function. The sequence grabber component then begins collecting data from all of the channels participating in the preview and plays that data appropriately. The sequence grabber component can pause and restart the preview by calling the `SGPause` function. The sequence grabber component uses the `SGStop` function to stop the preview. During the preview operation, the sequence grabber component calls your `SGIdle` function frequently, so that your channel can perform its operation.

4. When the application is done previewing, the sequence grabber component can start recording or close its connection to your component.

The following functions allow sequence grabber components to control your channel component:

- SGStartPreview
- SGStartRecord
- SGIdle
- SGUpdate
- SGStop
- SGWriteSamples
- SGPause
- SGPrepare
- SGRelease

Configuring Sequence Grabber Channel Components

This section discusses the functions that allow sequence grabber components to configure your channel component.

Sequence grabber components use a number of functions to establish the environment for grabbing or previewing digitized data. This section describes the channel component functions that allow the sequence grabber component to establish the environment for recording or previewing captured data.

The sequence grabber component uses the SGInitChannel function to initialize your channel prior to a record or preview operation.

The SGSetGWorld function allows the sequence grabber component to assign a graphics world to your component.

Configuration Functions for All Channel Components

Your channel is assigned to a sequence grabber component when the application calls the sequence grabber component's SGNewChannel function, described in the chapter Sequence Grabber Component Functions (page 49) in this document. The sequence grabber component must configure your channel before a preview or record operation. Your channel component must provide a number of functions that allow the sequence grabber to configure the characteristics of your channel. Several of these functions work on any channel component. This section discusses these general channel configuration functions.

In addition, channel components provide functions that are specific to the channel type. The sequence grabber component supplied by Apple uses two types of channel components: video channel components and sound channel components. See Configuration Functions for Video Channel Components (page 113) for information about the configuration functions that work only with video channels. See Configuration Functions for Sound Channel Components (page 114) for information about the configuration functions that work only with sound channels.

- The `SGSetChannelUsage` function specifies how your channel is to be used. The sequence grabber component can restrict a channel to use during record or preview operations. In addition, this function allows the sequence grabber component to specify whether your channel plays during a record operation. The `SGGetChannelUsage` function allows the sequence grabber component to determine a channel's usage.
- The `SGGetChannelInfo` function allows the sequence grabber component to determine some of the characteristics of your channel. For example, this function returns information indicating whether your channel has a visual or an audio representation.
- The `SGSetChannelPlayFlags` function lets the sequence grabber component influence the speed and quality with which your channel plays captured data. The `SGGetChannelPlayFlags` function allows the sequence grabber component to determine these flag settings.
- The `SGSetChannelMaxFrames` function establishes a limit on the number of frames that your channel component will capture from a channel.
- The `SGGetChannelMaxFrames` function enables the sequence grabber component to determine that limit.
- The `SGSetChannelRefCon` function allows the sequence grabber component to set the value of a reference constant that your component passes to its callback functions. See Using Callback Functions for Video Channel Components (page 116) for information about the callback functions that are supported by video channels.
- The `SGGetDataRate` function allows the sequence grabber component to determine how many bytes of captured data your channel is collecting each second.
- The `SGGetChannelSampleDescription` function allows the sequence grabber to retrieve your channel's sample description. The `SGGetChannelTimeScale` function allows it to obtain your channel's time scale.
- The sequence grabber can modify or retrieve your channel's clipping region by calling the `SGSetChannelClip` or `SGGetChannelClip` function, respectively. The sequence grabber can work with your channel's transformation matrix by calling the `SGSetChannelMatrix` and `SGGetChannelMatrix` functions.

Configuration Functions for Video Channel Components

Video channel components provide a number of functions that allow the sequence grabber to configure the channel's video characteristics. This section describes these video channel configuration functions, which the sequence grabber component uses only with video channels:

- The `SGSetChannelBounds` function allows the sequence grabber to set the display boundary rectangle for a video channel. The `SGGetChannelBounds` function determines a channel's boundary rectangle.
- The sequence grabber component uses the `SGGetSrcVideoBounds` function to determine the coordinates of the source video boundary rectangle. This rectangle defines the size of the source video image being captured by a video channel. The `SGSetVideoRect` function specifies a part of the source video boundary rectangle to be captured by the channel. The `SGGetVideoRect` function retrieves this active source video rectangle.
- Typically, video channel components use the Image Compression Manager to compress the video data they capture. The sequence grabber component can control many aspects of this image-compression process. The `SGSetVideoCompressorType` function specifies the type of image compressor to use. The sequence grabber can determine the type of image compressor currently in use by calling the

`SGGetVideoCompressorType` function. The sequence grabber component can specify a particular image compressor and set many image-compression parameters by calling the `SGSetVideoCompressor` function. The sequence grabber component can determine which image compressor is being used and its parameter settings by calling the `SGGetVideoCompressor` function.

- Video channel components typically work with a video digitizer component (see About Video Digitizer Components (page 125) for a discussion of video digitizer components). Sequence grabber components provide functions that allow an application to work with a channel's video digitizer component. Video channel components, in turn, must provide support for these functions. The sequence grabber component uses the `SGGetVideoDigitizerComponent` function to determine which video digitizer component is supplying data to your video channel component. The sequence grabber component sets a channel's video digitizer component by calling the `SGSetVideoDigitizerComponent` function. If an application changes any video digitizer settings by calling the video digitizer component directly, the sequence grabber component informs your video channel component by calling the `SGVideoDigitizerChanged` function.
- Some video source data may contain unacceptable levels of visual noise or artifacts. One technique for removing this noise is to capture the image and then reduce it in size. During the size reduction process, the noise can be filtered out. Some video channel components may provide functions that allow the sequence grabber component to filter the input video data. The `SGSetCompressBuffer` function sets a filter buffer for a video channel. The `SGGetCompressBuffer` function returns information about your filter buffer.
- The sequence grabber can work with a video channel's frame rate by calling the `SGSetFrameRate` and `SGGetFrameRate` functions. The sequence grabber can control whether your channel uses an offscreen buffer by calling your `SGSetUseScreenBuffer` and `SGGetUseScreenBuffer` functions.
- Your `SGAlignChannelRect` function allows the sequence grabber to determine a channel's optimum screen position.

Configuration Functions for Sound Channel Components

Sound channel components provide a number of functions that allow sequence grabber components to configure the component's sound channel. This section describes these sound channel configuration functions. The sequence grabber component uses these functions only with sound channels.

- The `SGSetChannelVolume` function allows the sequence grabber component to control a channel's sound volume. The sequence grabber component uses the `SGGetChannelVolume` function to determine a channel's volume.
- The `SGSetSoundInputDriver` specifies a channel's sound input device. The sequence grabber component can determine a channel's sound input device by calling the `SGGetSoundInputDriver` function. If an application changes any attributes of the sound input device, the sequence grabber component notifies your sound component by calling the `SGSoundInputDriverChanged` function.
- The sequence grabber component can control the amount of sound data your channel works with at one time by calling the `SGSetSoundRecordChunkSize` function. The sequence grabber component can determine this value by calling the `SGGetSoundRecordChunkSize` function.
- The sequence grabber component controls the rate at which your sound channel samples the input data by calling the `SGSetSoundInputRate` function. The sequence grabber component can determine the sample rate by calling the `SGGetSoundInputRate` function.
- The sequence grabber can control other sound input parameters by using your `SGSetSoundInputParameters` and `SGGetSoundInputParameters` functions.

Controlling Sequence Grabber Channel Components

Sequence grabber channel components must provide a full set of functions that allow the sequence grabber component to control the preview or record operation. The sequence grabber component can use these functions to start and stop the operation, to pause data collection, and to write captured data to a movie. This section describes these functions.

- The sequence grabber component uses the `SGStartPreview` function to start a preview operation. The `SGStartRecord` function starts a record operation. The `SGStop` function stops your channel component after a preview or record operation.
- The sequence grabber component grants processing time to your channel component by calling the `SGIdle` function. The sequence grabber notifies you of update events by calling your `SGUpdate` function.
- The sequence grabber pauses the current operation by calling the `SGPause` function.
- The sequence grabber component calls your `SGWriteSamples` function to write captured data to a movie file after a record operation.
- The sequence grabber component prepares your channel component for an upcoming preview or record operation by calling the `SGPrepare` function. This function also allows the sequence grabber component to verify that your component can support the parameters an application has specified. The `SGRelease` function releases system resources allocated during the `SGPrepare` function.

Recording

During a record operation, a sequence grabber component collects the data it captures and formats that data into a QuickTime movie. During a record operation, the sequence grabber component can also play the captured data for the user.

The following are the steps the sequence grabber component follows to record captured data.

1. As with a preview operation, the sequence grabber component establishes a connection to your channel component by calling the Component Manager's `OpenComponent` function. It then initializes your component by calling your `SGInitChannel` function.

2. The sequence grabber component then configures your component for the record operation. The `SGSetGWorld` function sets the graphics world in which the data is to be displayed. The `SGSetChannelUsage` function specifies each channel that is to be used for recording. At this time, the sequence grabber component can also specify whether your component is to play its data while recording. The application can then use the appropriate channel configuration functions to prepare your channel for the record operation. For video channels, it uses the functions discussed in Configuration Functions for Video Channel Components (page 113). For sound channels, the sequence grabber uses the functions discussed in Configuration Functions for Sound Channel Components (page 114).

3. The sequence grabber component starts the record operation by calling your `SGStartRecord` function. The sequence grabber component then begins collecting data from the channels it has assigned, stores the data in a QuickTime movie, and, optionally, plays that data appropriately. The sequence grabber can pause and restart the record process by calling the `SGPause` function. During the record operation, the sequence grabber component calls your `SGIdle` function frequently, so that your channel can perform

its operation. The sequence grabber component uses the `SGStop` function to stop the record operation. At this time, your component saves the movie in the appropriate movie file if the sequence grabber component instructs your component to do so by calling your `SGWriteSamples` function.

4. When the application is done recording, it either returns to previewing or closes its connection to your component.

Working With Callback Functions

Sequence grabber components provide callback functions that allow application developers to customize some aspects of capturing video data. It is your channel component's responsibility to call these callback functions at specified points in the data capture process. The application's function can then perform any special processing that is appropriate for the application. For example, an application can overlay text, such as a frame number, on each frame of video data as it is captured.

Note: Sound channel components do not support any callback functions.

Using Callback Functions for Video Channel Components

Sequence grabber components allow application developers to define a number of callback functions in their applications. Your channel component calls these functions at specific points in the process of collecting, compressing, and displaying the source visual data. By defining callback functions, a developer can control the process more precisely or customize the operation of the sequence grabber component and its channel components.

For example, you could use a callback function to draw a frame number on each video frame as it is collected. In this case, you could use either a compress callback function or a grab-complete callback function. You call the compress function after each frame is collected, in order to compress the frame. You call the grab-complete function just before the compress function or as soon as the frame has been captured.

Note that your channel component need not call each and every callback function. If some functions are inappropriate to the operation of your channel, do not call them. However, if your component calls one function of a pair, be sure to call the other. For example, if your component calls an application's grab function, you must also call its grab-complete function.

The sequence grabber component uses the `SGSetVideoBottlenecks` function to assign callback functions to your video channel. The `SGGetVideoBottlenecks` function allows the sequence grabber to determine the callback functions that have been assigned to your video channel. See the chapter Sequence Grabber Component Functions (page 49) in this document for details on `SGSetVideoBottlenecks` and `SGGetVideoBottlenecks`.

The following application-defined functions are supported by video channels and are described in Application-Defined Functions (page 65).

- `MyAddFrameFunction`
- `MyGrabCompressCompleteFunction`
- `MyCompressCompleteFunction`

- `MyGrabFunction`
- `MyCompressFunction`
- `MyTransferFrameFunction`
- `MyDisplayFunction`
- `MyGrabCompleteFunction`

Using Utility Functions for Video Channel Component Callback Functions

Sequence grabber components provide a number of functions that application-defined functions can use. Several channel functions support those sequence grabber component functions.

The sequence grabber component uses the `SGGetBufferInfo` function to obtain information about a buffer that contains data to be manipulated by a callback function. Application callback functions can use the `SGGetBufferInfo` function to obtain information about a buffer that you have passed. This information is valid only during record operations, or after your channel has been prepared to record. The `SGGetBufferInfo` function is described in the chapter Sequence Grabber Component Functions (page 49).

The following functions provide default behavior for application-defined grab, grab-complete, display, compress, compress-complete, add-frame, transfer-frame, display-compress, and grab-compress-complete functions:

- Your video channel component's `SGGrabFrame` function provides the default behavior for an application's grab function. Applications should call this function only from their grab function.
- Your channel component's `SGGrabFrameComplete` function provides the default behavior for an application's grab-complete function. Applications should call this function only from their grab-complete functions.
- Your channel component's `SGDisplayFrame` function provides the default behavior for an application's display function. Applications should call this function only from their display functions.
- Your video channel component's `SGCompressFrame` function provides the default behavior for an application's compress function. Applications should call this function only from their compress functions.
- Your channel component's `SGCompressFrameComplete` function provides the default behavior for an application's compress-complete function. Applications should call this function only from their compress-complete functions.
- Your component's `SGAddFrame` function provides the default behavior for an application's add-frame function. Applications should call this function only from their add-frame functions.
- Your component's `SGTransferFrameForCompress` function provides the default behavior for an application's transfer-frame function. Applications should call this function only from their transfer-frame functions.
- Your component's `SGGrabCompressComplete` function provides the default behavior for an application's grab-compress-complete function. Applications should call this function only from their grab-compress-complete function.
- Your component's `SGDisplayCompress` function provides the default behavior for an application's display-compress function. Applications should call this function only from their display-compress function.

Working With Channel Devices

Sequence grabbers provide a number of functions that allow applications to determine the devices that can be, or the device that is, attached to a given sequence grabber channel. These devices, in turn, allow the channel component to control the digitizing equipment. For example, video channels use video digitizer components, and sound channels use sound input drivers. Applications can use these functions to present a list of available devices to the user, allowing the user to select a specific device for each channel. The sequence grabber passes these functions on to your channel component.

The sequence grabber may use the `SGGetChannelDeviceList` function to retrieve a list of devices that may be used by your channel.

The sequence grabber can use the `SGSetChannelDevice` function to assign a device to your channel.

The `SGGetChannelDeviceList` function uses a device list structure to pass information about one or more channel devices. The `SGDeviceListRecord` data type defines the format of the device list structure.

```
typedef struct SGDeviceListRecord {
    short               count;                  /* count of devices */
    short               selectedIndex;          /* current device */
    long                reserved;               /* set to 0 */
    SGDeviceName        entry[1];               /* device names */
} SGDeviceListRecord, *SGDeviceListPtr, **SGDeviceList;
```

Field	Description
`count`	Indicates the number of devices described by this structure. The value of this field corresponds to the number of entries in the device name array defined by the `entry` field.
`selectedIndex`	Identifies the currently active device. The value of this field corresponds to the appropriate entry in the device name array defined by the `entry` field. Note that this value is 0-relative; that is, the first entry has an index number of 0, the second's value is 1, and so on.
`reserved`	Reserved for Apple. Always set to 0.
`entry`	Contains an array of device name structures. Each structure corresponds to one valid device. The `count` field indicates the number of entries in this array. The `SGDeviceName` data type defines the format of a device name structure; this data type is discussed next.

Device list structures contain an array of device name structures. Each device name structure identifies a single device that may be used by the channel. The `SGDeviceName` data type defines the format of a device name structure.

```
typedef struct SGDeviceName {
    Str63           name;                   /* device name */
    Handle          icon;                   /* device icon */
    long            flags;                  /* flags */
    long            refCon;                 /* set to 0 */
    long            reserved;               /* set to 0 */
} SGDeviceName;
```

Parameter	Description
name	Contains the name of the device. For video digitizer components, this field contains the component's name as specified in the component resource. For sound input drivers, this field contains the driver name.
icon	Contains a handle to the device's icon. Some devices may support an icon, which applications may choose to present to the user. If the device does not support an icon, or if the sequence grabber chooses not to retrieve this information (by setting the sgDeviceListWithIcons flag to 0 when it calls the SGGetChannelDeviceList function), set this field to nil.
flags	Reflects the current status of the device. The sgDeviceNameFlagDeviceUnavailable flag is defined. When set to 1, this flag indicates that this device is not currently available.
refCon	Reserved for Apple. Always set to 0.
reserved	Reserved for Apple. Always set to 0.

Utility Functions for Sequence Grabber Channel Components

This section describes several utility functions that sequence grabber components provide to sequence grabber channel components.

- The SGAddMovieData function lets you add data and sample references to a movie.
- Alternatively, you can use the SGWriteMovieData function to add data to a movie, and the SGAddFrameReference and SGGetNextFrameReference functions to keep track of sample references prior to creating a QuickTime movie from recorded data.
- The SGSortDeviceList function allows you to sort the entries in the device list that you create for the sequence grabber when it calls your SGGetChannelDeviceList function.
- The SGChangedSource function allows you to tell the sequence grabber that you have changed your source device.
- The SGAddFrameReference and SGGetNextFrameReference functions take a pointer to a frame information structure as a parameter. The SeqGrabFrameInfo data type defines the format of a frame information structure.

```
struct SeqGrabFrameInfo {
    long            frameOffset;        /* offset to the sample */
    long            frameTime;          /* time that frame was captured */
    long            frameSize;          /* number of bytes in sample */
    SGChannel       frameChannel;       /* current connection to channel */
    long            frameRefCon;        /* reference constant for channel */
};
```

Field	Description
frameOffset	Specifies the offset to the sample. Your channel component obtains this value from the SGWriteMovieData function.

Field	Description
`frameTime`	Specifies the time at which your channel component captured the frame. This time value is relative to the data sequence. That is, this time is not represented in the context of any fixed time scale. Rather, your channel component must choose and use a time scale consistently for all sample references.
`frameSize`	Specifies the number of bytes in the sample described by the sample reference.
`frameChannel`	Identifies the current connection to your channel.
`frameRefCon`	Contains a reference constant for use by your channel component. You can use this value in any way that is appropriate for your channel component. For example, video channel components may use this value to store a reference to frame differencing information for a temporally compressed image sequence.

Text Channel Components

Text channel components are a type of sequence grabber channel component. A text channel component captures text for use in QuickTime movies. It is controlled by a sequence grabber component. Applications programmers will normally interact with the higher-level sequence grabber component, and do not need to read this chapter.

Text channel components make use of text digitizer components, which digitze text from particular sources, such as the closed-caption text from a video input. Text channel components abstract this level of detail, allowing the sequence grabber component to work with a stream of text without regard to its source.

You should read this chapter if you are developing a text channel component, a text digitizer, or a sequence grabber component.

About the QuickTime 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.

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.

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.

Text Channel Component Functions

The following channel component functions are unique to text channel components. These functions allow captured text to be formatted prior to being previewed or added to a movie.

- `SGSetFontName`
- `SGSetFontSize`
- `SGSetTextForeColor`
- `SGSetTextBackColor`

- SGSetJustification
- SGGetTextReturnToSpaceValue
- SGSetTextReturnToSpaceValue

The QuickTime text channel component also supports some, but not all, functions defined for sequence grabber channel components and sequence grabber panel components. The supported functions are the following:

- General sequence grabber component functions:
  - SGSetGWorld
  - SGNewChannel
  - SGStartPreview
  - SGStartRecord
  - SGIdle
  - SGStop
  - SGPause
  - SGPrepare
  - SGRelease
  - SGGetChannelDeviceList
  - 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

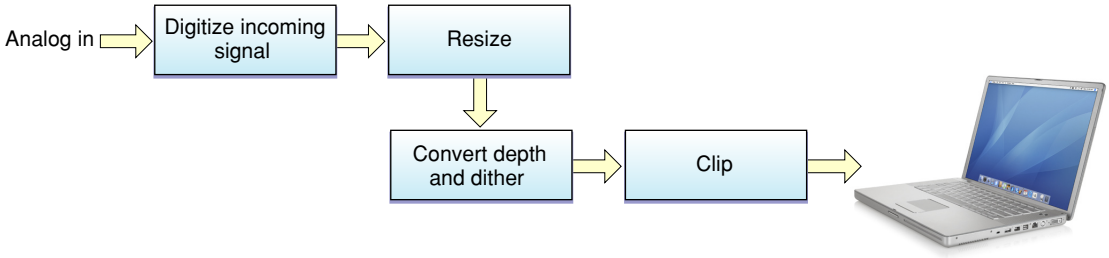
For further information about these functions, see Sequence Grabber Component Functions (page 49).

About Video Digitizer Components

Video digitizer components convert video input into a digitized color image that is compatible with the graphics system of a computer. For example, a video digitizer may convert input analog video into a specified digital format. The input may be any video format and type, whereas the output must be intelligible to the computer's display system. Once the digitizer has converted the input signal to an appropriate digital format, it then prepares the image for display by resizing the image, performing necessary color conversions, and clipping to the output window. At the end of this process, the digitizer component places the converted image into a buffer you specify. If that buffer is the current frame buffer, the image appears on the user's computer screen.

Analog-to-Digital Conversion

Figure 8-1 shows the steps involved in converting the analog video signal to digital format and preparing the digital data for display. Some video digitizer components perform all these steps in hardware. Others perform some or all of these steps in software. Others may perform only a few of these steps, in which case it is up to the program that is using the video digitizer to perform the remaining tasks.



Video digitizer components resize the image by applying a transformation matrix to the digitized image. Your application specifies the matrix that is applied to the image. Matrix operations can enlarge or shrink an image, distort the image, or move the location of an image. The Movie Toolbox provides a set of functions that make it easy for you to work with transformation matrices.

Before the digitized image can be displayed on your computer, the video digitizer component must convert the image into an appropriate color representation. This conversion may involve dithering or pixel depth conversion. The digitizer component handles this conversion based on the destination characteristics you specify.

Video digitizer components may support clipping. Digitizers that do support clipping can display the resulting image in regions of arbitrary shapes. See Clipping (page 140) for a discussion of the techniques that digitizer components can use to perform clipping.

Types of Video Digitizer Components

Video digitizer components fall into four categories, distinguished by their support for clipping a digitized video image:

- basic digitizers, which do not support clipping
- alpha channel digitizers, which clip by means of an alpha channel
- mask plane digitizers, which clip by means of a mask plane
- key color digitizers, which clip by means of key colors

Basic video digitizer components are capable of placing the digitized video into memory, but they do not support any graphics overlay or video blending. If you want to perform these operations, you must do so in your application. For example, you can stop the digitizer after each frame and do the work necessary to blend the digitized video with a graphics image that is already being displayed. Unfortunately, this may cause jerkiness or discontinuity in the video stream. Other types of digitizers that support clipping make this operation much easier for your application.

Alpha channel digitizer components use a portion of each display pixel to represent the blending of video and graphical image data. This part of each pixel is referred to as an **alpha channel.** The size of the alpha channel differs depending upon the number of bits used to represent each pixel. For 32 bits per pixel modes, the alpha channel is represented in the 8 high-order bits of each 32-bit pixel. These 8 bits can define up to 256 levels of blend. For 16 bits per pixel modes, the alpha channel is represented in the high-order bit of the pixel and defines one level of blend (on or off).

Mask plane digitizer components use a pixel map to define blending. Values in this mask correspond to pixels on the screen, and they define the level of blend between video and graphical image data.

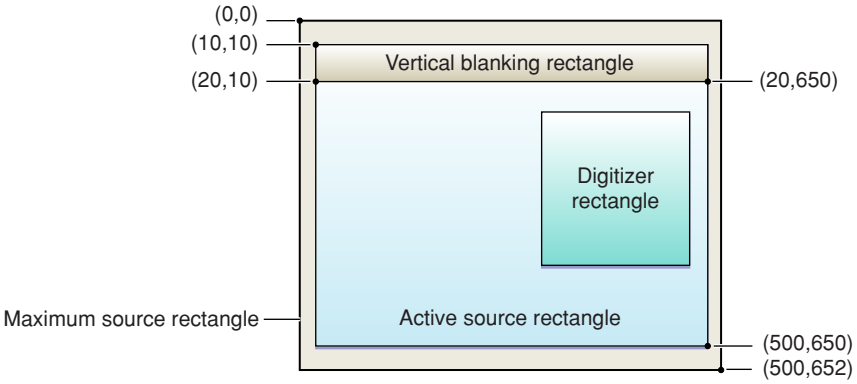
Key color digitizer components determine where to display video data based upon the color currently being displayed on the output device. These digitizers reserve one or more colors in the color table; these colors define where to display video. For example, if blue is reserved as the key color, the digitizer replaces all blue pixels in the display rectangle with the corresponding pixels of video from the input video source.

Source Coordinate Systems

Your application can control what part of the source video image is extracted. The digitizer then converts the specified portion of the source video signal into a digital format for your use. Video digitizer components define four areas you may need to manipulate when you define the source image for a given operation. These areas are

- the maximum source rectangle
- the active source rectangle
- the vertical blanking rectangle
- the digitizer rectangle

Figure 8-2 shows the relationships between these rectangles.



The **maximum source rectangle** defines the maximum source area that the digitizer component can grab. This rectangle usually encompasses both the vertical and horizontal blanking areas. The **active source rectangle** defines that portion of the maximum source rectangle that contains active video. The **vertical blanking rectangle** defines that 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 might not be contained in the maximum source rectangle.

You specify the **digitizer rectangle,** which defines that portion of the active source rectangle that you want to capture and convert.

Using Video Digitizer Components

This section describes how you can control a video digitizer component. It is divided into the following topics:

- Specifying Destinations (page 127) discusses how you tell the digitizer where to put the converted video data.
- Starting and Stopping the Digitizer (page 128) discusses how you control digitization.
- Multiple Buffering (page 129) describes a technique for improving performance.
- Obtaining an Accurate Time of Frame Capture (page 129) tells how the sequence grabber usually supplies video digitizers with a time base. This time base lets your application get an accurate time for the capture of any specified frame.

Specifying Destinations

Video digitizer components provide several functions that allow applications to specify the destination for the digitized video stream produced by the digitizer component. You have two options for specifying the destination for the video data stream in your application.

- The first option requires that the video be digitized as RGB pixels and placed into a destination pixel map. This option allows the video to be placed either onscreen or offscreen, depending upon the placement of the pixel map. Your application can use the `VDSetPlayThruDestination` function to set the characteristics for this option. Your application can use the `VDPreflightDestination` function to determine the capabilities of the digitizer. All video digitizer components must support this option.

- The second option uses a global boundary rectangle to define the destination for the video. This option always results in onscreen images and is useful with digitizers that support hardware direct memory access (DMA) across multiple screens. The digitizer component is responsible for any required color depth conversions, image clipping and resizing, and so on. Your application can use the `VDSetPlayThruGlobalRect` function to set the characteristics for this option. Your application can use the `VDPreflightGlobalRect` function to determine the capabilities of the digitizer. Not all video digitizer components support this option.

Setting Video Destinations

Video digitizer components provide several functions that allow applications to specify the destination for the digitized video stream produced by the digitizer component. Applications have two options for specifying the destination for the video data stream:

- The first option requires that the video be digitized as RGB pixels and placed into a destination pixel map. This option allows the video to be placed either onscreen or offscreen, depending upon the placement of the pixel map. You can use the `VDSetPlayThruDestination` function in your application to set the characteristics for this option. The `VDPreflightDestination` function lets you determine the capabilities of the digitizer in your application. All video digitizer components must support this option. The `VDGetPlayThruDestination` function lets you get data about the current video destination.
- The second option uses a global boundary rectangle to define the destination for the video. This option is useful only with digitizers that support hardware DMA. You can use the `VDSetPlayThruGlobalRect` function in your application to set the characteristics for this option. You can use the `VDPreflightGlobalRect` function in your application to determine the capabilities of the digitizer. Not all video digitizer components support this option.

The `VDGetMaxAuxBuffer` function returns information about a buffer that may be located on some special hardware.

Starting and Stopping the Digitizer

You can control digitization on a frame-by-frame basis in your application. The `VDGrabOneFrame` function digitizes a single video frame. All video digitizer components support this function.

Alternatively, you can use the `VDSetPlayThruOnOff` function to enable or disable digitization. When digitization is enabled, the video digitizer component places video into the specified destination continuously. The application stops the digitizer by disabling digitization. This function can be used with both destination options. However, not all video digitizer components support this function.

Controlling Digitization

This section describes the video digitizer component functions that allow applications to control video digitization.

Video digitizer components allow applications to start and stop the digitizing process. Your application can request continuous digitization or single-frame digitization. When a digitizer component is operating continuously, it automatically places successive frames of digitized video into the specified destination. When a digitizer component works with a single frame at a time, the application and other software, such as an image compressor component, control the speed at which the digitized video is processed.

You can use the `VDSetPlayThruOnOff` function in your application to enable or disable digitization. When digitization is enabled, the video digitizer component places digitized video frame into the specified destination continuously. The application stops the digitizer by disabling digitization. This function can be used with both destination options.

Alternatively, your application can control digitization on a frame-by-frame basis. The `VDGrabOneFrame` and `VDGrabOneFrameAsync` functions digitize a single video frame; `VDGrabOneFrame` works synchronously, returning control to your application when it has obtained a complete frame, while `VDGrabOneFrameAsync` works asynchronously. The `VDDone` function helps you to determine when the `VDGrabOneFrameAsync` function is finished with a video frame. Your application can define the buffers for use with asynchronous digitization by calling the `VDSetupBuffers` function. Free the buffers by calling the `VDReleaseAsyncBuffers` function.

The `VDSetFrameRate` function allows applications to control the digitizer's frame rate. The `VDGetDataRate` function returns the digitizer's current data rate.

Multiple Buffering

You can improve the performance of frame-by-frame digitization by using multiple destination buffers for the digitized video. Your application defines a number of destination buffers to the video digitizer component and specifies the order in which those buffers are to be used. The digitizer component then fills the buffers, allowing you to switch between the buffers more quickly than your application otherwise could. In this manner, you can grab a video sequence at a higher rate with less chance of data loss. This technique can be used with both destination options.

You define the buffers to the digitizer by calling the `VDSetupBuffers` function. The `VDGrabOneFrameAsync` function starts the process of grabbing a single video frame. The `VDDone` function allows you to determine when the digitizer component has finished a given frame.

Obtaining an Accurate Time of Frame Capture

The sequence grabber typically gives video digitizers a time base so your application can obtain an accurate time for the capture of any given frame. Applications can set the digitizer's time base by calling the `VDSetTimeBase` function.

Controlling Compressed Source Devices

Some video digitizer components may provide functions that allow applications to work with digitizing devices that can provide compressed image data directly. Such devices allow applications to retrieve compressed image data without using the Image Compression Manager. However, in order to display images from the compressed data stream, there must be an appropriate decompressor component available to decompress the image data.

Video digitizers that can support compressed source devices set the `digiOutDoesCompress` flag to 1 in their capability flags. See Capability Flags (page 141) for more information about these flags.

Applications can use the `VDGetCompressionTypes` function to determine the image-compression capabilities of a video digitizer. The `VDSetCompression` function allows applications to set some parameters that govern image compression.

Applications control digitization by calling the `VDCompressOneFrameAsync` function, which instructs the video digitizer to create one frame of compressed image data. The `VDCompressDone` function returns that frame. When an application is done with a frame, it calls the `VDReleaseCompressBuffer` function to free the buffer. An application can force the digitizer to place a key frame into the sequence by calling the `VDResetCompressSequence` function. Applications can turn compression on and off by calling `VDSetCompressionOnOff`.

Applications can obtain the digitizer's image description structure by calling the `VDGetImageDescription` function. Applications can set the digitizer's time base by calling the `VDSetTimeBase` function.

All of the digitizing functions described in this section support only asynchronous digitization. That is, the video digitizer works independently to digitize each frame. Applications are free to perform other work while the digitizer works on each frame.

The video digitizer component manages its own buffer pool for use with these functions. In this respect, these functions differ from the other video digitizer functions that support asynchronous digitization. See Controlling Digitization (page 128) for more information about these functions.

Creating Video Digitizer Components

Video digitizer components are the most convenient mechanism for presenting new sources of video data to QuickTime. For example, if you are developing special-purpose video hardware that digitizes video images from a previously unsupported source device, you should create a video digitizer component so that applications or sequence grabber components can obtain data from your device.

Video digitizer components support a rich functional interface that can accommodate devices with quite varied capabilities. To relieve you from having to support irrelevant functions, Apple has made several video digitizer functions optional. For information about the functions your digitizer component must support, see Required Functions (page 131). For information about other functions, see Optional Functions (page 132).

Required Functions

At a minimum, your video digitizer component must support the following functions:

- `VDGetActiveSrcRect`
- `VDGetCurrentFlags`
- `VDGetDigitizerInfo`
- `VDGetDigitizerRect`
- `VDGetFieldPreference`
- `VDGetInput`
- `VDGetInputFormat`
- `VDGetMaxSrcRect`
- `VDGetNumberOfInputs`
- `VDGetPlayThruDestination`
- `VDGetVBlankRect`
- `VDGetVideoDefaults`
- `VDGrabOneFrame`
- `VDPreflightDestination`
- `VDSetDigitizerRect`
- `VDSetFieldPreference`
- `VDSetInput`
- `VDSetInputStandard`
- `VDSetPlayThruDestination`

- `VDGetCompressionTime`
- `VDSetDataRate`

All of these functions are required for all video digitizer components.

Optional Functions

Based on the type of device your component supports, you may have to implement functions other than those listed in Required Functions (page 131) and you may have to set some of your component's capability flags. Read this section to learn which additional functions your component needs to support and how to set your capability flags properly.

If your component does not support a particular function, be sure to return a result code value of `digiUnimpErr`.

Note: Hardware support for the simultaneous capture and display of frames on the screen is called *playthrough* in these sections.

Frame Grabbers Without Playthrough

Suppose your video digitization hardware grabs frames but cannot simultaneously display the frames on the screen. Suppose also that your hardware supplies the grabbed frames in QuickDraw pixel maps at specific pixel depths (say, 16 and 32 bits per pixel).

In this case, you should set the following component capability flags:

Flag	Setting
`digiOutDoes16`	Set this flag to 1.
`digiOutDoes32`	Set this flag to 1. Set other depth flags to 0.
`digiOutDoesHWPlayThru`	Set this flag to 0.
`digiOutDoesDMA`	Set this flag to 0.

If your component can operate asynchronously, you should also set the following flag:

Flag	Setting
`digiOutDoesAsyncGrabs`	Set this flag to 1 if your component can operate asynchronously.

Frame grabbers that support asynchronous operation must support the following optional functions:

- `VDDone`
- `VDGrabOneFrameAsync`

- `VDReleaseAsyncBuffers`
- `VDSetupBuffers`

Frame Grabbers With Hardware Playthrough

If your frame grabber hardware provides support for playing the captured images directly, you need to support one additional function beyond those discussed in Frame Grabbers Without Playthrough (page 132). The `VDSetPlayThruOnOff` function allows the application to turn playthrough on and off.

You should also set the `digiOutDoesHWPlayThru` capability flag to 1. In addition, be sure to use the `gdh` field in the digitizer information structure to identify your component's display device. For details on the video digitizer information structure, see The Digitizer Information Structure (page 146) .

Key Color and Alpha Channel Devices

As a further elaboration on a basic frame grabber, your device could support the display or mixing of output data via an alpha channel or through the use of key colors (see Types of Video Digitizer Components (page 126) for more information about alpha channels and key colors). In either case, image data cannot be read directly from the screen. Therefore, you must set the `digiOutDoesUnreadableScreenBits` capability flag to 1. For more on the video digitizer capability flags, see Capability Flags (page 141) .

Your component must load its alpha channel or fill in the key color whenever playthrough is enabled or when the destination changes.

Compressed Source Devices

You may create a video digitizer component that supports a device that delivers compressed image data. In this case, your component is not capable of displaying the data directly.

Your component should set the following capability flags:

Flag	Setting
`digiOutDoesCompress`	Set this flag to 1.
`digiOutDoesCompressOnly`	Set this flag to 1 if your component cannot display the images directly.
`digiOutDoesPlayThruDuringCompress`	Set this flag to 1 if your component cannot display the images directly.

In addition, frame grabbers that support compressed source devices must support the following optional functions:

- `VDCompressDone`
- `VDCompressOneFrameAsync`
- `VDGetCompressionTypes`

- VDGetDataRate
- VDGetImageDescription
- VDResetCompressSequence
- VDSetCompression
- VDSetCompressionOnOff
- VDSetFrameRate
- VDSetTimeBase

If your hardware generates compressed data that cannot be decompressed by any standard QuickTime image decompressor components, be sure to provide an appropriate decompressor component so that the data you provide can be displayed.

Function Request Codes

You can use the following enumerators to refer to the request codes for each of the functions that your component must support.

```
enum {
    kSelectVDGetMaxSrcRect          = 0x1,  /* VDGetMaxSrcRect (required) */
    kSelectVDGetActiveSrcRect       = 0x2,  /* VDGetActiveSrcRect
                                               (required) */
    kSelectVDSetDigitizerRect       = 0x3,  /* VDSetDigitizerRect
                                               (required) */
    kSelectVDGetDigitizerRect       = 0x4,  /* VDGetDigitizerRect
                                               (required) */
    kSelectVDGetVBlankRect          = 0x5,  /* VDGetVBlankRect (required) */
    kSelectVDGetMaskPixMap              = 0x6,  /* VDGetMaskPixMap */
    kSelectVDGetPlayThruDestination     = 0x8,  /* VDGetPlayThruDestination
                                               (required) */
    kSelectVDUseThisCLUT                = 0x9,  /* VDUseThisCLUT */
    kSelectVDSetInputGammaValue         = 0xA,  /* VDSetInputGammaValue */
    kSelectVDGetInputGammaValue         = 0xB,  /* VDGetInputGammaValue */
    kSelectVDSetBrightness              = 0xC,  /* VDSetBrightness */
    kSelectVDGetBrightness              = 0xD,  /* VDGetBrightness */
    kSelectVDSetContrast                = 0xE,  /* VDSetContrast */
    kSelectVDSetHue                     = 0xF,  /* VDSetHue */
    kSelectVDSetSharpness               = 0x10, /* VDSetSharpness */
    kSelectVDSetSaturation              = 0x11, /* VDSetSaturation */
    kSelectVDGetContrast                = 0x12, /* VDGetContrast */
    kSelectVDGetHue                     = 0x13, /* VDGetHue */
    kSelectVDGetSharpness               = 0x14,/* VDGetSharpness */
    kSelectVDGetSaturation              = 0x15, /* VDGetSaturation */
    kSelectVDGrabOneFrame               = 0x16, /* VDGrabOneFrame
                                               (required) */
    kSelectVDGetMaxAuxBuffer            = 0x17, /* VDGetMaxAuxBuffer */
    kSelectVDGetDigitizerInfo           = 0x19, /* VDGetDigitizerInfo
                                               (required) */
    kSelectVDGetCurrentFlags            = 0x1A, /* VDGetCurrentFlags
                                               (required) */
    kSelectVDSetKeyColor                = 0x1B, /* VDSetKeyColor */
```

```
    kSelectVDGetKeyColor                = 0x1C, /* VDGetKeyColor */
    kSelectVDAddKeyColor                = 0x1D, /* VDAddKeyColor */
    kSelectVDGetNextKeyColor            = 0x1E, /* VDGetNextKeyColor */
    kSelectVDSetKeyColorRange           = 0x1F, /* VDSetKeyColorRange */
    kSelectVDGetKeyColorRange           = 0x20, /* VDGetKeyColorRange */
    kSelectVDSetDigitizerUserInterrupt  = 0x21,
                                        /* VDSetDigitizerUserInterrupt */
    kSelectVDSetInputColorSpaceMode     = 0x22, /* VDSetInputColorSpaceMode */
    kSelectVDGetInputColorSpaceMode     = 0x23, /* VDGetInputColorSpaceMode */
    kSelectVDSetClipState               = 0x24, /* VDSetClipState */
    kSelectVDSetClipState               = 0x25, /* VDGetClipState */
    kSelectVDSetClipRgn                 = 0x26, /* VDSetClipRgn */
    kSelectVDClearClipRgn               = 0x27, /* VDClearClipRgn */
    kSelectVDGetCLUTInUse               = 0x28, /* VDGetCLUTInUse */
    kSelectVDSetPLLFilterType           = 0x29, /* VDSetPLLFilterType */
    kSelectVDGetPLLFilterType           = 0x2A, /* VDGetPLLFilterType */
    kSelectVDGetMaskandValue            = 0x2B, /* VDGetMaskandValue */
    kSelectVDSetMasterBlendLevel        = 0x2C, /* VDSetMasterBlendLevel */
    kSelectVDSetPlayThruDestination     = 0x2D, /* VDSetPlayThruDestination */
    kSelectVDSetPlayThruOnOff           = 0x2E, /* VDSetPlayThruOnOff */
    kSelectVDSetFieldPreference         = 0x2F, /* VDSetFieldPreference
                                                   (required) */
    kSelectVDGetFieldPreference         = 0x30, /* VDGetFieldPreference
                                                   (required) */
    kSelectVDPreflightDestination       = 0x32, /* VDPreflightDestination
                                                   (required) */
    kSelectVDPreflightGlobalRect        = 0x33, /* VDPreflightGlobalRect */
    kSelectVDSetPlayThruGlobalRect      = 0x34, /* VDSetPlayThruGlobalRect */
    kSelectVDSetInputGammaRecord        = 0x35, /* VDSetInputGammaRecord */
    kSelectVDGetInputGammaRecord        = 0x36, /* VDGetInputGammaRecord */
    kSelectVDSetBlackLevelValue         = 0x37,/* VDSetBlackLevelValue */
    kSelectVDGetBlackLevelValue         = 0x38, /* VDGetBlackLevelValue */
    kSelectVDSetWhiteLevelValue         = 0x39, /* VDSetWhiteLevelValue */
    kSelectVDGetWhiteLevelValue         = 0x3A, /* VDGetWhiteLevelValue */
    kSelectVDGetVideoDefaults           = 0x3B, /* VDGetVideoDefaults */
    kSelectVDGetNumberOfInputs          = 0x3C,/* VDGetNumberOfInputs */
    kSelectVDGetInputFormat             = 0x3D, /* VDGetInputFormat */
    kSelectVDSetInput                   = 0x3E, /* VDSetInput */
    kSelectVDGetInput                   = 0x3F, /* VDGetInput */
    kSelectVDSetInputStandard           = 0x40, /* VDSetInputStandard */
    kSelectVDSetupBuffers               = 0x41, /* VDSetupBuffers */
    kSelectVDGrabOneFrameAsync          = 0x42, /* VDGrabOneFrameAsync */
    kSelectVDDone                       = 0x43, /* VDDone */
    kSelectVDSetCompression             = 0x44, /* VDSetCompression */
    kSelectVDCompressOneFrameAsync      = 0x45, /* VDCompressOneFrameAsync */
    kSelectVDCompressDone               = 0x46, /* VDCompressDone */
    kSelectVDReleaseCompressBuffer      = 0x47, /* VDReleaseCompressBuffer */
    kSelectVDGetImageDescription        = 0x48, /* VDGetImageDescription */
    kSelectVDResetCompressSequence      = 0x49, /* VDResetCompressSequence */
    kSelectVDSetCompressionOnOff        = 0x4A, /* VDSetCompressionOnOff */
    kSelectVDGetCompressionTypes        = 0x4B, /* VDGetCompressionTypes */
    kSelectVDSetTimeBase                = 0x4C, /* VDSetTimeBase */
    kSelectVDSetFrameRate               = 0x4D, /* VDSetFrameRate */
    kSelectVDGetDataRate                = 0x4E, /* VDGetDataRate */
    kSelectVDGetSoundInputDriver        = 0x4F, /* VDGetSoundInputDriver */
    kSelectVDGetDMADepths               = 0x50, /* VDGetDMADepths */
    kSelectVDGetPreferredTimeScale      = 0x51, /* VDGetPreferredTimeScale */
    kSelectVDReleaseAsyncBuffers        = 0x52,/* VDReleaseAsyncBuffers */
```

```
};
```

Video Digitizer Component API

Introduction

This chapter describes the functions that are provided by video digitizer components. These functions are described from the perspective of an application that uses video digitizer components. If you are developing a video digitizer component, your digitizer component must behave as described here.

These functions specify the video digitizer components for their requests with a reference obtained from the Component Manager's `OpenComponent` or `OpenAComponent` function.

Note: If you are developing an application that uses video digitizer components, you should read the sections that are appropriate to your application. If you are developing a video digitizer component, you should read all the sections.

Component Type and Subtype Values

Apple has defined a type value for video digitizer components. All video digitizer components have a component type value of `'vdig'`. You can use the following constant to specify the component type value.

```
#define videoDigitizerComponentType = 'vdig'
```

There are no special conventions applied to the subtype value of video digitizer components.

Getting Information About Video Digitizer Components

You can use the `VDGetDigitizerInfo` function in your application to retrieve information about the capabilities of a video digitizer component. You can use the `VDGetCurrentFlags` function to obtain current status information from a video digitizer component.

Setting Source Characteristics

This section discusses the video digitizer component functions that allow applications to set the spatial characteristics of the source video signal. You can use these functions in your application to set and retrieve information about the maximum source rectangle, the active source rectangle, the vertical blanking rectangle, and the digitizer rectangle. For a complete discussion of the relationship between these rectangles, see About Video Digitizer Components (page 125).

You can use the `VDGetMaxSrcRect` function in your application to get the size and location of the maximum source rectangle. Similarly, the `VDGetActiveSrcRect` function allows you to get this information about the active source rectangle, and the `VDGetVBlankRect` function enables you to obtain information about the vertical blanking rectangle.

You can use the `VDSetDigitizerRect` function to set the size and location of the digitizer rectangle. The `VDGetDigitizerRect` function lets you retrieve the size and location of this rectangle.

Selecting an Input Source

This section discusses the video digitizer component functions that allow applications to select an input video source.

Some of these functions provide information about the available video inputs. Applications can use the `VDGetNumberOfInputs` function to determine the number of video inputs supported by the digitizer component. The `VDGetInputFormat` function allows applications to find out the video format (composite, s-video, or component) employed by a specified input.

You can use the `VDSetInput` function in your application to specify the input to be used by the digitizer component. The `VDGetInput` function returns the currently selected input.

The `VDSetInputStandard` function allows you to specify the video signaling standard to be used by the video digitizer component.

Controlling Color

Video digitizer components support color digitization. Therefore, these components provide several functions that allow applications to control the color digitization process.

You can use `VDSetInputColorSpaceMode` in your application to enable and disable color digitization; you can use the `VDGetInputColorSpaceMode` function to determine whether color digitization is enabled. The `VDUseThisCLUT` function allows you to specify a color lookup table to be used by the video digitizer component. In cases where the component cannot accommodate a particular lookup table, your application can use the `VDGetCLUTInUse` function to retrieve the color lookup table used by the digitizer component.

Your application can determine whether a digitizer component supports color digitization by examining the input capability flags of the component. Specifically, if the `digiInDoesColor` flag is set to 1, the component supports color digitization. Applications can use the `VDGetCurrentFlags` function to obtain the input capability flags of a component. See Getting Information About Video Digitizer Components (page 137) for more information.

Your application can determine a digitizer's supported pixel depths by calling the `VDGetDMADepths` function.

Controlling Analog Video

Some video digitizer components may provide functions that allow applications to control the characteristics of the input analog video signal. This section describes these analog video functions.

The `VDGetVideoDefaults` function returns the suggested default values for the analog video parameters that can be affected by functions described in this section.

A number of functions affect gamma correction. The `VDSetInputGammaRecord` and `VDGetInputGammaRecord` functions work with gamma structures. You can use the `VDSetInputGammaValue` and `VDGetInputGammaValue` functions to allow your application to set particular gamma values.

The `VDSetBlackLevelValue`, `VDGetBlackLevelValue`, `VDSetWhiteLevelValue`, and `VDGetWhiteLevelValue` functions allow applications to work with black levels and white levels in the source video. **Black level** refers to the degree of blackness in an image. This is a common setting on a video digitizer. 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 is a significant setting because it can be adjusted so that there is little or no noise in an image. **White level** refers to the degree of whiteness in an image. It is also a common video digitizer setting.

The `VDSetContrast`, `VDGetContrast`, `VDSetSharpness`, and `VDGetSharpness` functions allow applications to work with contrast and sharpness values in the source video. The `VDGetBrightness` and `VDSetBrightness` functions allow applications to work with the image brightness setting.

The `VDSetHue`, `VDGetHue`, `VDSetSaturation`, and `VDGetSaturation` functions allow applications to work with hue and saturation settings in the source video.

Selectively Displaying Video

Video digitizer components may support one of three methods of selectively displaying video on the computer screen. The three methods are key colors, alpha channels, and blend masks. For a complete description of these techniques for selectively displaying video, see About Video Digitizer Components (page 125).

Your application can determine whether a video digitizer component supports selective video display by examining the component's digitizer information structure. Specifically, the `vdigType` field indicates the type of blending supported by the digitizer. Applications can use the `VDGetDigitizerInfo` function to retrieve a component's digitizer information structure.

Some video digitizer components support the use of key colors as a mechanism for selectively displaying video on the computer screen. When a key color is active, the digitizer component replaces all screen occurrences of that color with the appropriate portion of the source video. Video digitizer components that support key colors provide a number of functions to applications. Those functions are described in this section.

Your applications can use the `VDSetKeyColor`, `VDAddKeyColor`, and `VDSetKeyColorRange` functions to set one or more key colors for a video digitizer component. The `VDGetKeyColor`, `VDGetNextKeyColor`, and `VDGetKeyColorRange` functions allow your application to retrieve information about the currently active key colors.

Alpha channels and blend masks work similarly to one another. Digitizer components that support alpha channels use a portion of each pixel value to indicate the degree of video display for that pixel. Digitizer components that support blend masks use the mask to indicate the degree of video display for corresponding pixels.

Your applications can use the `VDGetMaskandValue` function to determine the appropriate mask value for a desired blend level. The `VDSetMasterBlendLevel` function allows applications to set a blend level that applies to the entire source video image. The `VDGetMaskPixMap` function allows applications to retrieve the pixel map that defines the blend mask.

Clipping

Some video digitizer components can clip the output video image based on an arbitrary clipping region. Your application can determine whether a video digitizer component supports clipping by examining the digitizer information structure of the component. Specifically, if the `digiOutDoesMask` flag is set to 1 in the `outputCapabilityFlags` field of the appropriate digitizer information structure, the component supports clipping. See The Digitizer Information Structure (page 146) for details. Your application can obtain a component's digitizer information structure by calling the `VDGetDigitizerInfo` function. This section describes the functions provided to applications by components that support clipping.

Applications can use the `VDSetClipState` and `VDGetClipState` functions to enable and disable clipping, and to determine whether clipping is enabled. Applications can use the `VDSetClipRgn` and `VDClearClipRgn` functions to manipulate the clipping region. Applications can use these functions only during an active grab sequence. Applications set the initial clipping settings by calling either `VDSetPlayThruDestination` or `VDSetPlayThruGlobalRect`.

Note: The functions that manipulate clipping and clipping state operate on a clipping region in addition to the one specified by the mask passed by the `VDSetPlayThruDestination` and `VDSetUpBuffers` functions. To determine the final clipping regions, intersect these two clippings.

Utility Functions

A number of utility functions may be supported by some video digitizer components:

- The `VDSetPLLFilterType` and `VDGetPLLFilterType` functions allow applications to control which **phase-locked loop (PLL)** is used by a video digitizer component that supports multiple PLLs.
- The `VDSetFieldPreference` and `VDGetFieldPreference` functions allow applications to control which field is used for some vertical scaling operations.
- The `VDSetDigitizerUserInterrupt` function allows applications to install custom interrupt functions that are called by the video digitizer component.
- The `VDGetSoundInputDriver` function allows an application to retrieve information about a digitizer's sound input driver.
- The `VDGetPreferredTimeScale` function allows an application to determine a digitizer's preferred time scale.
- 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.
- Video digitizers may return timecode information for an incoming video signal by responding to `VDGetTimeCode`.
- You can use the `VDGetInputFormat` function to find out the video format employed by a specified input.

Application-Defined Function

Applications can provide a custom interrupt function in the `userInterruptProc` parameter of the `VDSetDigitizerUserInterrupt` function. Every custom interrupt function must support the following interface:

```
pascal void MyInterruptProc (long flags, long refcon);
```

The `flags` parameter indicates when the interrupt function has been called. The video digitizer component sets these flags to indicate the circumstances in which the function has been called. The following flags are defined:

Flag	Description
`Bit 0`	Even-line field interrupt. If this flag is set to 1, the video digitizer component is about to display an even-line field.
`Bit 1`	Odd-line field interrupt. If this flag is set to 1, the video digitizer component is about to display an odd-line field.

The `refcon` parameter contains parameter data that is appropriate for the interrupt function. The application assigns the value of the reference constant when it sets the interrupt function.

Capability Flags

Video digitizer components report their capabilities to your application by means of capability flags. These flags are formatted as part of the digitizer information structure you obtain by calling the `VDGetDigitizerInfo` function. There are two sets of flags: one set describes the input capabilities of the video digitizer component, and the other describes its output capabilities.

Video digitizer components support the following input capability flags:

Flag	Description
`digiInDoesNTSC`	Indicates that the video digitizer supports National Television System Committee (NTSC) format input video signals. This flag is set to 1 if the digitizer component supports NTSC video.

Flag	Description
digiInDoesPAL	Indicates that the video digitizer component supports Phase Alternation Line (PAL) format input video signals. This flag is set to 1 if the digitizer component supports PAL video.
digiInDoesSECAM	Indicates that the video digitizer component supports Systeme Electronique Couleur avec Memoire (SECAM) format input video signals. This flag is set to 1 if the digitizer component supports SECAM video.
digiInDoesGenLock	Indicates that the video digitizer component supports genlock; that is, the digitizer can derive its timing from an external time base. This flag is set to 1 if the digitizer component supports genlock.
digiInDoesComposite	Indicates that the video digitizer component supports composite input video. This flag is set to 1 if the digitizer component supports composite input.
digitInDoesSVideo	Indicates that the video digitizer component supports s-video input video. This flag is set to 1 if the digitizer component supports s-video input.
digiInDoesComponent	Indicates that the video digitizer component supports RGB input video. This flag is set to 1 if the digitizer component supports RGB input.
digiInVTR_Broadcast	Indicates that the video digitizer component can distinguish between an input signal that emanates from a videotape player and a broadcast signal. This flag is set to 1 if the digitizer component can differentiate between the two different signal types.
digiInDoesColor	Indicates that the video digitizer component supports color input. This flag is set to 1 if the digitizer component can accept color input.
digiInDoesBW	Indicates that the video digitizer component supports grayscale input. This flag is set to 1 if the digitizer component can accept grayscale input.

Video digitizer components support the following output capability flags:

Flag	Description
digiOutDoes1	Indicates that the video digitizer component can work with pixel maps that contain 1-bit pixels. If this flag is set to 1, then the digitizer component can write images that contain 1-bit pixels. If this flag is set to 0, then the digitizer component cannot handle such images.
digiOutDoes2	Indicates that the video digitizer component can work with pixel maps that contain 2-bit pixels. If this flag is set to 1, then the digitizer component can write images that contain 2-bit pixels. If this flag is set to 0, then the digitizer component cannot handle such images.
digiOutDoes4	Indicates that the video digitizer component can work with pixel maps that contain 4-bit pixels. If this flag is set to 1, then the digitizer component can write images that contain 4-bit pixels. If this flag is set to 0, then the digitizer component cannot handle such images.

Flag	Description
digiOutDoes8	Indicates that the video digitizer component can work with pixel maps that contain 8-bit pixels. If this flag is set to 1, then the digitizer component can write images that contain 8-bit pixels. If this flag is set to 0, then the digitizer component cannot handle such images.
digiOutDoes16	Indicates that the video digitizer component can work with pixel maps that contain 16-bit pixels. If this flag is set to 1, then the digitizer component can write images that contain 16-bit pixels. If this flag is set to 0, then the digitizer component cannot handle such images.
digiOutDoes32	Indicates that the video digitizer component can work with pixel maps that contain 32-bit pixels. If this flag is set to 1, then the digitizer component can write images that contain 32-bit pixels. If this flag is set to 0, then the digitizer component cannot handle such images.
digiOutDoesDither	Indicates that the video digitizer component supports dithering. If this flag is set to 1, the component supports dithering of colors. If this flag is set to 0, the digitizer component does not support dithering.
digiOutDoesStretch	Indicates that the video digitizer component can stretch images to arbitrary sizes. If this flag is set to 1, the digitizer component can stretch images. If this flag is set to 0, the digitizer component does not support stretching.
digiOutDoesShrink	Indicates that the video digitizer component can shrink images to arbitrary sizes. If this flag is set to 1, the digitizer component can shrink images. If this flag is set to 0, the digitizer component does not support shrinking.
digiOutDoesMask	Indicates that the video digitizer component can handle clipping regions. If this flag is set to 1, the digitizer component can mask to an arbitrary clipping region. If this flag is set to 0, the digitizer component does not support clipping regions.
digiOutDoesDouble	Indicates that the video digitizer component supports stretching to quadruple size when displaying the output video. The parameters for the stretch operation are specified in the matrix structure for the request; the component modifies the scaling attributes of the matrix. If this flag is set to 1, the digitizer component can stretch an image to exactly four times its original size, up to the maximum size specified by the maxDestHeight and maxDestWidth fields in the digitizer information structure. If this flag is set to 0, the digitizer component does not support stretching to quadruple size.
digiOutDoesQuad	Indicates that the video digitizer component supports stretching an image to 16 times its original size when displaying the output video. The parameters for the stretch operation are specified in the matrix structure for the request; the component modifies the scaling attributes of the matrix. If this flag is set to 1, the digitizer component can stretch an image to exactly 16 times its original size, up to the maximum size specified by the maxDestHeight and maxDestWidth fields in the digitizer information structure. If this flag is set to 0, the digitizer component does not support this capability.

Flag	Description
`digiOutDoesQuarter`	Indicates that the video digitizer component can shrink an image to one-quarter of its original size when displaying the output video. The parameters for the shrink operation are specified in the matrix structure for the request; the component modifies the scaling attributes of the matrix. If this flag is set to 1, the digitizer component can shrink an image to exactly one-quarter of its original size, down to the minimum size specified by the `minDestHeight` and `minDestWidth` fields in the digitizer information structure. If this flag is set to 0, the digitizer component does not support this capability.
`digiOutDoesSixteenth`	Indicates that the video digitizer component can shrink an image to 1/16 of its original size when displaying the output video. The parameters for the shrink operation are specified in the matrix structure for the request; the digitizer component modifies the scaling attributes of the matrix. If this flag is set to 1, the digitizer component can shrink an image to exactly 1/16 of its original size, down to the minimum size specified by the `minDestHeight` and `minDestWidth` fields in the digitizer information structure. If this flag is set to 0, the digitizer component does not support this capability.
`digiOutDoesRotate`	Indicates that the video digitizer component can rotate an image when displaying the output video. The parameters for the rotation are specified in the matrix structure for an operation. If this flag is set to 1, the digitizer component can rotate the image. If this flag is set to 0, the digitizer component cannot rotate the resulting image.
`digiOutDoesHorizFlip`	Indicates that the video digitizer component can flip an image horizontally when displaying the output video. The parameters for the horizontal flip are specified in the matrix structure for an operation. If this flag is set to 1, the digitizer component can flip the image. If this flag is set to 0, the digitizer component cannot flip the resulting image.
`digiOutDoesVertFlip`	Indicates that the video digitizer component can flip an image vertically when displaying the output video. The parameters for the vertical flip are specified in the matrix structure for an operation. If this flag is set to 1, the digitizer component can flip the image. If this flag is set to 0, the digitizer component cannot flip the resulting image.
`digiOutDoesSkew`	Indicates that the video digitizer component can skew an image when displaying the output video. Skewing an image distorts it linearly along only a single axis; for example, drawing a rectangular image into a parallelogram-shaped region. The parameters for the skew operation are specified in the matrix structure for the request. If this flag is set to 1, the digitizer component can skew an image. If this flag is set to 0, the digitizer component does not support this capability.
`digiOutDoesBlend`	Indicates that the video digitizer component can blend the resulting image with a matte when displaying the output video. The matte is provided by the application by defining either an alpha channel or a mask plane. If this flag is set to 1, the digitizer component can blend. If this flag is set to 0, the digitizer component does not support this capability.

Flag	Description
digiOutDoesWarp	Indicates that the video digitizer component can warp an image when displaying the output video. Warping an image distorts it along one or more axes, perhaps nonlinearly, in effect "bending" the result region. The parameters for the warp operation are specified in the matrix structure for the request. If this flag is set to 1, the digitizer component can warp an image. If this flag is set to 0, the digitizer component does not support this capability.
digiOutDoesDMA	Indicates that the video digitizer component can write to any screen or to offscreen memory. If this flag is set to 1, the digitizer component can use DMA to write to any screen or memory location.
digiOutDoes-HWPlayThru	Indicates that the video digitizer component does not need idle time in order to display its video. If this flag is set to 1, your application does not need to grant processor time to the digitizer component at normal display speeds.
digiOutDoesILUT	Indicates that the video digitizer component supports inverse lookup tables for indexed color modes. If this flag is set to 1, the digitizer component uses inverse lookup tables when appropriate.
digiOutDoesKeyColor	Indicates that the video digitizer component supports clipping by means of key colors. If this flag is set to 1, the digitizer component can clip to a region defined by a key color.
digiOutDoes-AsyncGrabs	Indicates that the video digitizer component can operate asynchronously. If this flag is set to 1, your application can use the VDSetupBuffers and VDGrabOneFrameAsync functions.
digiOutDoes-UnreadableScreenBits	Indicates that the video digitizer may place pixels on the screen that cannot be used when compressing images.
digiOutDoesCompress	Indicates that the video digitizer component supports compressed source devices. These devices provide compressed data directly, without having to use the Image Compression Manager.
digiOutDoesCompress-Only	Indicates that the video digitizer component only provides compressed image data; the component cannot provide displayable data. This flag only applies to digitizers that support compressed source devices.
digiOutDoesPlayThru-DuringCompress	Indicates that the video digitizer component can draw images on the screen at the same time that it is delivering compressed image data. This flag only applies to digitizers that support compressed source devices.

Data Types

This section discusses the data structures that are used by video digitizer components and by applications that use video digitizer components.

The Digitizer Information Structure

Your application can retrieve information about the capabilities and current status of a video digitizer component. You call the `VDGetDigitizerInfo` function to retrieve all this information from a video digitizer component. In response, the component formats a digitizer information structure. The contents of this structure fully define the capabilities and current status of the video digitizer component.

Note: If you are interested only in the current status information, you can call the `VDGetCurrentFlags` function. This function returns the input and output current flags of the video digitizer component.

The `DigitizerInfo` data type defines the layout of the digitizer information structure:

```
struct DigitizerInfo {
   short       vdigType;
   long        inputCapabilityFlags;
   long        outputCapabilityFlags;
   long        inputCurrentFlags;
   long        outputCurrentFlags;
   short       slot;
   GDHandle    gdh;
   GDHandle    maskgdh;
   short       minDestHeight;
   short       minDestWidth;
   short       maxDestHeight;
   short       maxDestWidth;
   short       blendLevels;
   long        reserved;
   };
```

Field	Description
`vdigType`	Specifies the type of video digitizer component. Valid values are listed below.
`inputCapabilityFlags`	Specifies the capabilities of the video digitizer component with respect to the input video signal.
`outputCapabilityFlags`	Specifies the capabilities of the video digitizer component with respect to the output digitized video information.
`inputCurrentFlags`	Specifies the current status of the video digitizer with respect to the input video signal.
`outputCurrentFlags`	Specifies the current status of the video digitizer with respect to the output digitized video information.
`slot`	Identifies the slot that contains the video digitizer interface card.
`gdh`	Contains a handle to the graphics device that defines the screen to which the digitized data is to be written. Set this field to `nil` if your application is not constrained to a particular graphics device.
`maskgdh`	Contains a handle to the graphics device that contains the mask plane. This field is used only by digitizers that clip by means of mask planes.

Field	Description
minDestHeight	Indicates the smallest height value the digitizer component can accommodate in its destination.
minDestWidth	Indicates the smallest width value the digitizer component can accommodate in its destination.
maxDestHeight	Indicates the largest height value the digitizer component can accommodate in its destination.
maxDestWidth	Indicates the largest width value the digitizer component can accommodate in its destination.
blendLevels	Specifies the number of blend levels the video digitizer component supports.
reserved	Reserved. Set this field to 0.

The inputCapabilityFlags and outputCapabilityFlags values are listed in Capability Flags (page 141).

The vdigType field may contain these values:

Constant	Description
vdTypeBasic	Basic video digitizer; does not support any clipping
vdTypeAlpha	Supports clipping by means of an alpha channel
vdTypeMask	Supports clipping by means of a mask plane
vdTypeKey	Supports clipping by means of key colors

The Buffer List Structure

If you are using more than one asynchronous output buffer, you must define the output buffers to the video digitizer component. You define these output buffers by calling the VDSetupBuffers function. You specify the buffers to that function in a buffer list structure. Note that all the output buffers must be the same size and must accommodate output rectangles of the same dimensions.

The VdigBufferRecList data type defines a buffer list structure.

```
struct VdigBufferRecList {
    short                   count;
    MatrixRecordPtr         matrix;
    RgnHandle               mask;
    VdigBufferRec           list[1];
};
```

Field	Description
`count`	Specifies the number of buffers defined by this structure. The value of this field must correspond to the number of entries in the `list` array.
`matrix`	Specifies the transformation matrix that is applied to all of the destination rectangles before the video image is displayed. You must specify a matrix. If you do not want to perform any transformations, use the identity matrix.
`mask`	Specifies a clipping region that is applied to the destination rectangle before the video image is displayed. Note that this region applies to only the first destination buffer. If you want the region to apply to all of your destination buffers, you must do this yourself. If you do not want to specify a clipping region, set this field to `nil`.
`list`	Contains an array of output buffer specifications. Each buffer is represented by a buffer structure. The format and content of this structure are described in the next section.

The Buffer Structure

The `VdigBufferRec` data type defines a buffer structure.

```
typedef struct {
    PixMapHandle        dest;
    Point               location;
    long                reserved;
} VdigBufferRec;
```

Field	Description
`dest`	Contains a handle to the pixel map that defines the destination buffer.
`location`	Specifies the location of the video destination in the pixel map specified by the `dest` field. This point identifies the upper-left corner of the destination rectangle. The size and scaling of the destination rectangle are governed by the `matrix` and `mask` fields of the buffer list structure that contains this structure.
`reserved`	Reserved for use by Apple. Set this field to 0.

Document Revision History

This table describes the changes to *QuickTime Movie Creation Guide*.

Date	Notes
2007-01-08	Revised artwork
2006-01-10	New document that describes how to create a QuickTime movie from within an application.
	Replaces "Movie Toolbox: Creating Movies," "Sequence Grabber Components," "Sequence Grabber Channel Components," "Sequence Grabber Panel Components," "Text Channel Components," and "Video Digitizer Components."
2002-09-17	New document that describes the QuickTime functions that an application can use to construct movies.

Document Revision History