QuickTime Media Types and Media Handlers Guide

QuickTime > Media Types & Media Handlers

2006-01-10

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

Contents

Figures and Listings

Introduction to QuickTime Media Types and Media Handlers Guide

This book introduces the idea of QuickTime media handler components and provides details of the video, sound, text, timecode, and tween media handlers.

> **Note:** This book replaces five previously separate Apple documents: "Media Handlers: Introduction, Video and Sound," "Text Media Handler, "Time Code Media Handler, "Tween Components and Tween Media," and "Tween Media Handler."

The last half of this book describes the media handler components that perform tween operations, sometimes called tweeners. It also describes tween operations performed by QuickTime for tween types that are native to QuickTime.

A tween operation lets you algorithmically generate an output value for any point in a time interval. The input for a tween is a small number of values, often as few as one or two, from which a range of values can be derived. You can use output from a tween either to modify tracks in a QuickTime movie or to perform actions unrelated to movies.

For a general overview of media handler technology in QuickTime, read About Media Handlers (page 11). The rest of this book is of interest primarily to developers who need to develop new media handlers for QuickTime. You need to read the last five chapters of this book if you are a developer planning to work with or create QuickTime tween components.

Organization of This Document

This book is divided into nine chapters:

- About Media Handlers (page 11) describes media handlers, components that are responsible for interpreting and manipulating a media's sample data.
- Video and Sound Media Handlers (page 13) describes the media handlers that interpret and manipulate video data.
- Text Media Handlers (page 15) describes media handlers that you can use to add plain or styled text samples to a movie, indicate scrolling and highlighting properties for the text, search for text, and highlight specified text runs.
- Timecode Media Handlers (page 19) describe media handlers that let QuickTime movies store timing information derived from a movie's original source material, such as SMPTE timecodes.
- Tweens and Tween Operations (page 25) introduces tweens and their uses, and provides an overview of the tween operations that are possible.
- Using Tween Components (page 35) describes how to create tween containers that the tween media handler uses.

- Creating a Tween Component (page 73) explains how to create a tween component for a new data type, a new interpolation algorithm, or both.
- Tween Components and Native Tween Types (page 57) describes the native tween types handled by QuickTime; the tween components included in QuickTime; and the constants, data types, and routines associated with tween components.
- Tween Media Handlers (page 29) describes media handlers that are used to send tween values from a tween track to a receiving track, such as a video track or a sound track.

See Also

For a discussion of QuickTime movie time management, see *QuickTime Movie Internals Guide*.

The following Apple books cover aspects of QuickTime programming related to media handlers:

- *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.
- *QuickTime Compression and Decompression Guide* introduces you to the QuickTime Image Compression Manager and its associated components, which provide image-compression and image-decompression services to applications and to other QuickTime components.
- *QuickTime Video Effects and Transitions Guide* tells you how to program QuickTime video effects and transitions between movie tracks and graphic images.
- *QuickTime Component Creation Guide* tells you how to build new components to extend the capabilities of QuickTime, including media handlers and preview components.
- *QuickTime API Reference* provides encyclopedic details of all the functions, callbacks, data types and structures, atom types, and constants in the QuickTime API.

About Media Handlers

The Movie Toolbox does not contain direct support for manipulating specific media types. This work is performed by media handler components. Media handlers are components that are responsible for interpreting and manipulating a media's sample data.

Each media type has its own media handler, which deals with the specific characterisics of that media type. Apple provides media handlers for video, sound, text, sprites, timecodes, tweens, and QuickTime music. Because media handlers are implemented as components, new media handlers can be created to support new media types, or to add new features to the handling of existing media.

Applications do not normally interact with media handlers directly; applications make calls to the Movie Toolbox, which calls media handlers as needed.

Selecting Media Handlers

Media handler components are responsible for interpreting and manipulating a media's sample data. Each type of media has its own media handler, which deals with the specific characteristics of the media data. The Movie Toolbox provides a set of functions that allow you to gather information about a media handler and assign a particular media handler to a media. This section discusses those functions.

Each media handler has an associated data handler for each data reference. The data handler is responsible for fetching, storing, and caching the data that the media handler uses. The Movie Toolbox provides functions that allow you to get information about data handlers and to assign a particular data handler to a media.

Media Selection Functions

Media handler selection uses the following functions:

- The `GetMediaHandler` and `GetMediaHandlerDescription` functions allow you to retrieve information about a media handler.
- You can use the `SetMediaHandler` function to assign a media handler to a media.
- The `GetMediaDataHandler` and `GetMediaDataHandlerDescription` functions enable you to retrieve information about a data handler. Use the `SetMediaDataHandler` function to assign a data handler to a media.

Media Property Functions

QuickTime provides two functions, `GetMediaPropertyAtom` and `SetMediaPropertyAtom`, for setting and retrieving the property atom container of a media handler. This allows you to get and set the properties of a track associated with the specified media handler.

Video and Sound Media Handlers

Video media handlers are responsible for interpreting and manipulating video data. These media handlers allow you to call them directly to work with some graphics settings. This section lists the functions supported by video media handlers.

Video media handlers maintain a graphics mode and color value that affect the display of video data. You can use the `SetVideoMediaGraphicsMode` and `GetVideoMediaGraphicsMode` functions to work with these characteristics.

Sample descriptions for video media are stored in image description structures.

Sound media handlers are responsible for interpreting and manipulating sound data. These media handlers allow you to call them directly to work with some audio settings.

Sound media handlers maintain balance information for their audio data. You can use the `SetSoundMediaBalance` and `GetSoundMediaBalance` functions to work with a handler's balance setting.

Sample descriptions for sound media are stored in sound description structures.

Video Media Handler Functions

The following functions can be used specifically with video media handler components:

- `SetVideoMediaGraphicsMode`
- `GetVideoMediaGraphicsMode`

The Image Description Structure

Sample descriptions for video media are stored in image description structures. An image description structure contains information that defines the characteristics of a compressed image or sequence. Data in the image description structure indicates the type of compression that was used, the size of the image when displayed, the resolution at which the image was captured, and so on. One image description structure may be associated with one or more compressed frames.

The `ImageDescription` data type defines the layout of an image description structure. In addition, an image description structure may contain additional data in extensions and custom color tables. The Image Compression Manager provides functions that allow you to get and set the data in image description structure extensions and custom color tables.

Sound Media Handler Functions

Two functions can be used with sound media handler components:

- `SetSoundMediaBalance`
- `GetSoundMediaBalance`

The Sound Description Structure

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

The `SoundDescription` data type defines the layout of a sound description structure.

Text Media Handlers

Applications do not normally interact with media handlers directly; applications make calls to the Movie Toolbox, which calls media handlers as needed. The material in this chapter will be primarily of interest to developers whose applications will allow the creation or editing of text tracks in QuickTime movies.

Text Operations

You can use text media handlers to

- add plain or styled text samples to a movie
- indicate scrolling and highlighting properties for the text
- search for text
- highlight specified text

A particular text sample has a default font, size, typeface, and color as well as a location (text box) within the track bounds to be drawn. The data format allows you to include style run information for the text. You can set flags to clip the display to the text box, inhibit automatic scaling of text as the track bounds are scaled, scroll the text, and specify if text is to be displayed at all.

The Movie Toolbox provides functions to help you add text samples to a track. You can use the `TextMediaAddTextSample` function to add text to a media. The `TextMediaAddTESample` function allows you to specify a `TextEdit` handle (which may have multiple style runs) to be added to a media. The `TextMediaAddHiliteSample` function allows you to indicate highlighting for text that has just been added with the `TextMediaAddTextSample` or `TextMediaAddTESample` function.

The format of the text data that is added to the media is a 16-bit length word followed by the text. The length word specifies the number of bytes in the text. Optionally, one or more atoms of additional data may follow. An atom is structured as a 32-bit length word followed by a 32-bit type followed by some data. The length word includes the size of the data as well as the length and type fields (in other words, the size of the data plus 8).

Text atom types include the `style` atom (`'styl'`), the shrunken text box atom (`'tbox'`), the highlighting atom (`'hlit'`), the scroll delay atom (`'dlay'`), and the highlight color atom (`'hclr'`).

The format of the `style` atom is the same as TextEdit's `StScrpRec` data type. A `StScrpRec` data type is a short integer specifying the number of style runs followed by that number of `ScrpSTElement` data types, each specifying a different style run.

The shrunken text box atom is added when you set the `dfShrinkTextBoxToFit` display flag (in the `TextMediaAddTextSample` or `TextMediaAddTESample` function). Its format is simply the rectangle of the shrunken box (16 bytes total, including length and type).

The highlighting atom is added if the hiliteStart and hiliteEnd parameters are set appropriately in the TextMediaAddTextSample or TextMediaAddTESample function. When TextMediaAddHiliteSample is called, an empty text sample (the first 2 bytes are 0) with a highlighting atom is added to the media. The format is two long integers indicating the start and end of the highlighting (16 bytes total).

The scroll delay atom specifies the scroll delay for a sample. It is a long value that specifies the delay time. It consists of 12 bytes, including the length and type fields.

The highlight color atom specifies the highlight color for a sample. Its format is an RGBColor data type (that is, 2 bytes red, 2 bytes green, and 2 bytes blue). It consists of 14 bytes, including the length and type fields.

Text Media Handler Functions

The following functions can be used with text media handler components. They support such tasks as adding plain or styled text samples to a movie, setting scrolling and highlighting, and text searching.

- TextMediaSetTextSampleData
- TextMediaAddTextSample
- TextMediaAddTESample
- TextMediaAddHiliteSample
- TextMediaFindNextText
- TextMediaHiliteTextSample
- TextMediaSetTextProc

The TextMediaAddTextSample, TextMediaAddTESample, and TextMediaAddHiliteSample functions convert text into the text media format and add it to the media. To use these functions, you need to:

- create a text track and media
- call the BeginMediaEdits function
- call the TextMediaAddTextSample, TextMediaAddTESample, or TextMediaAddHiliteSample function, as appropriate
- call the EndMediaEdits function
- call the InsertMediaIntoTrack function

Movie import and export components help to get common data types (such as 'PICT' or 'snd ') into and out of movies easily. The text import component allows you to get text into a movie using the following principles:

- If you try to paste text, the text is inserted at the current position. The text import component tries to find an existing text track that fits the text.
- If no text tracks exist and there is an insertion operation, the newly created text track has the same position and size as the movie box.
- If there is an addition operation (using the Shift key), the new track is added below the movie at a height that fits the text.

- If a text track exists but the text does not fit, a new text track with sufficient height to accommodate the text is created in the same location as the existing one.
- If you hold down the Option key when you paste, the text is added in parallel at some default duration.
- If you hold down both the Option and Shift keys, the duration of the text is determined by the length of the current selection.
- If style information is on the Clipboard, it is used; otherwise, the text appears in the default 12-point application font, centered, in white on a black background.

If you want more control over how the text is added (for example, if you want to set some display flags or a new track position), your application must:

- intercept the text paste
- instantiate its own text import component using the component type `'eat '` and component subtype `'TEXT'`
- use functions including `MovieImportSetSampleDuration`, `MovieImportSetSampleDescription`, `MovieImportSetDimensions`, and `MovieImportSetAuxilliaryData` (with `'styl'` and a `StScrpHandle` data type)
- call the `MovieImportHandle` function with the text data
- adjust the location of the track, if desired (since the text import component may place it below the movie box)

The Movie Toolbox provides functions that allow you to search for and highlight text. You can use the `TextMediaFindNextText` function to search for text in a text track, and the `TextMediaHiliteTextSample` function to highlight specified text in a text track.

You can use the `TextMediaSetTextProc` function to specify a customized function whenever a new text sample is added to a movie.

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

Timecode Media Handlers

Timecode media handlers allow QuickTime movies to store timing information derived from the movie's original source material, such as SMPTE timecodes, as distinct from the time base data which is a part of any QuickTime movie.

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

Timecode Tracks

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

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

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

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

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

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

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

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

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

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

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

Timecode Samples

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

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

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

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

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

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

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

Term	Definition
flags	Contains flags that provide some timecode format information.

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

The following flags are defined in the flags parameter:

Term	Definition
tcDropFrame	Indicates that the timecode "drops" frames occasionally in order to stay in synchronization. Some timecodes run at other than a whole number of frames per second. For example, NTSC video runs at 29.97 frames per second. In order to resynchronize between the timecode rate and a 30 frames-per-second playback rate, the timecode drops a frame at a predictable time (in much the same way that leap years keep the calendar synchronized). Set this flag to 1 if the timecode uses the dropframe technique.
tc24HourMax	Indicates that the timecode values wrap at 24 hours. Set this flag to 1 if the timecode hour value wraps (that is, returns to 0) at 24 hours.
tcNegTimesOK	Indicates that the timecode supports negative time values. Set this flag to 1 if the timecode allows negative values.
tcCounter	Indicates that the timecode should be interpreted as a simple counter, rather than as a time value. This allows the timecode to contain either time information or counter (such as a tape counter) information. Set this flag to 1 if the timecode contains counter information.

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

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

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

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

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

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

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

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

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

Timecode Media Handler Functions

The following functions are specific to the timecode media handler:

- `TCGetCurrentTimeCode`
- `TCGetTimeCodeAtTime`
- `TCTimeCodeToFrameNumber`
- `TCFrameNumberToTimeCode`
- `TCTimeCodeToString`
- `TCSetSourceRef`
- `TCGetSourceRef`
- `TCSetTimeCodeFlags`
- `TCGetTimeCodeFlags`
- `TCSetDisplayOptions`
- `TCGetDisplayOptions`

Tweens and Tween Operations

Every **tween** operation is based on a collection of one or more values from which a range of output values can be algorithmically derived. Each tween is assigned a time duration, and an output value can be generated for any time value within the duration. In the simplest kind of tween operation, a pair of values is provided as input and values between the two values are generated as output.

For example, if the tween data is a pair of integers, 0 and 5, the duration of the tween operation is 100, and the algorithm used to generate output values is linear interpolation (in which generated values, when graphed, fall on a straight line between the input values), the output returned for a time value of 0 is 0, the output for 25 is 1.25, the output for 50 is 2.5, and the output for 100 is 5.

QuickTime supports a variety of **tween types.** Each tween type is distinguished from other types by these characteristics:

- Input values or structures of a particular type.
- A particular number of input values or structures (most often one or two).
- Output values or structures of a particular type.
- A particular algorithm used to derive the output values.

Tween operations for each tween type are performed by a tween component that is specific to that type or, for a number of tween types that are native to QuickTime, by QuickTime itself. Movies and applications that use tweens do not need to specify the tween component to use; QuickTime identifies a tween type by its tween type identifier and automatically routes its data to the correct tween component or to QuickTime. If you need to perform tween operations that QuickTime does not support, you can develop a new tween component, as described in Creating a Tween Component (page 73).

When a movie contains a tween track, the tween media handler invokes the necessary component (or built-in QuickTime code) for tween operations and delivers the results to another media handler. The receiving media handler can then use the values it receives to modify its playback. For example, the data in a tween track can be used to alter the volume of a sound track.

Tweens can also be used outside of movies by applications or other software that can use the values they generate.

Tween Types

Each of the tween types supported by QuickTime belongs to one of these categories:

- **Numeric** tween types, which have pairs of numeric values, such as long integers, as input. For these types, linear interpolation is used to generate output values.

- The **polygon** tween type, which takes three four-sided polygons as input. One polygon (such as the bounds for a sprite or track) is transformed, and the two others specify the start and end of the range of polygons into which the tween operation maps it. You can use the output (a `MatrixRecord` data structure) to map the source polygon into any intermediate polygon. The intermediate polygon is interpolated from the start and end polygons for each particular time in the tween duration.
- **Path** tween types have as input a QuickTime vector data stream for a path. (For information about QuickTime vectors, see Tween Components and Native Tween Types (page 57)). Four of the path tween types also have as input a percentage of path's length; for these types, either a point on the path or a `MatrixRecord` data structure is returned. Two other path tween types treat the path as a function: one returns the `y` value of the point on the path with a given `x` value, and the other returns the `x` value of the point on the path with a given `y` value.
- The **list** tween type has as input a QT atom container that contains leaf atoms of a specified atom type. For this tween type, the duration of the tween operation is divided by the number of leaf atoms of the specified type. For time points within the first time division, the data for the first leaf atom is returned; for the second time division, the data for the second leaf atom is returned; and so on. The resulting tween operation proceeds in discrete steps (one step for each leaf atom), instead of the relatively continuous tweening produced by other tween types.

Single Tweens and Tween Sequences

A tween operation can include one or more tweens. A tween operation that includes just one tween is called a **single tween.** For a single tween, results for any time points in the tween duration are derived from that tween. A **tween sequence** contains more than one tween of the same type. To specify when each tween is used, you specify an ending percentage for each tween. For example, if you have a tween sequence containing three tweens and want to use the first tween for the first quarter of the tween duration, the second tween for the second quarter of the tween duration, and the third tween for the remainder of the tween duration, you set the end percentage for the first tween to .25, for the second to .5, and for the third to 1.0. The first tween in the sequence always begins at the beginning, and each subsequent tween begins after the end percentage of the tween before it.

Interpolation Tweens

Interpolation tweens are tweens that modify other tweens. The output of an interpolation tween must be a time value, and the time values generated are used in place of the input time values of the tween being modified. For example, you can use a path tween whose data specifies a curve to modify a tween that uses linear interpolation for its algorithm. The starting and ending values for the modified tween remain the same, but the rate at which output values change over time is determined by the shape of the curve.

Once you create an interpolation tween, you can use it to modify any number of other tweens. You can do this by specifying maximum and/or minimum output values of the interpolation tween to match the time values for the tween to be modified. For example, if there is a curve whose shape describes the natural decay rate for several different sounds, you can can define a single interpolation tween for that curve and apply it, with appropriate maximum and minimum values, to all of the sounds.

An interpolation tween can modify another interpolation tween; the only requirement is that the output of each interpolation tween must be a time value. The ability to define series of interpolations makes it possible to create libraries of standard modifications that can be used together to generate more complex transformations.

Tween Media Handlers

A **tween track** is a special track in a movie that is used exclusively as a modifier track. The data it contains, known as **tween data,** is used to generate values that modify the playback of other tracks, usually by interpolating values. The tween media handler sends these values to other media handlers; it never presents data. For an introduction to modifier tracks, see *QuickTime Movie Internals Guide*.

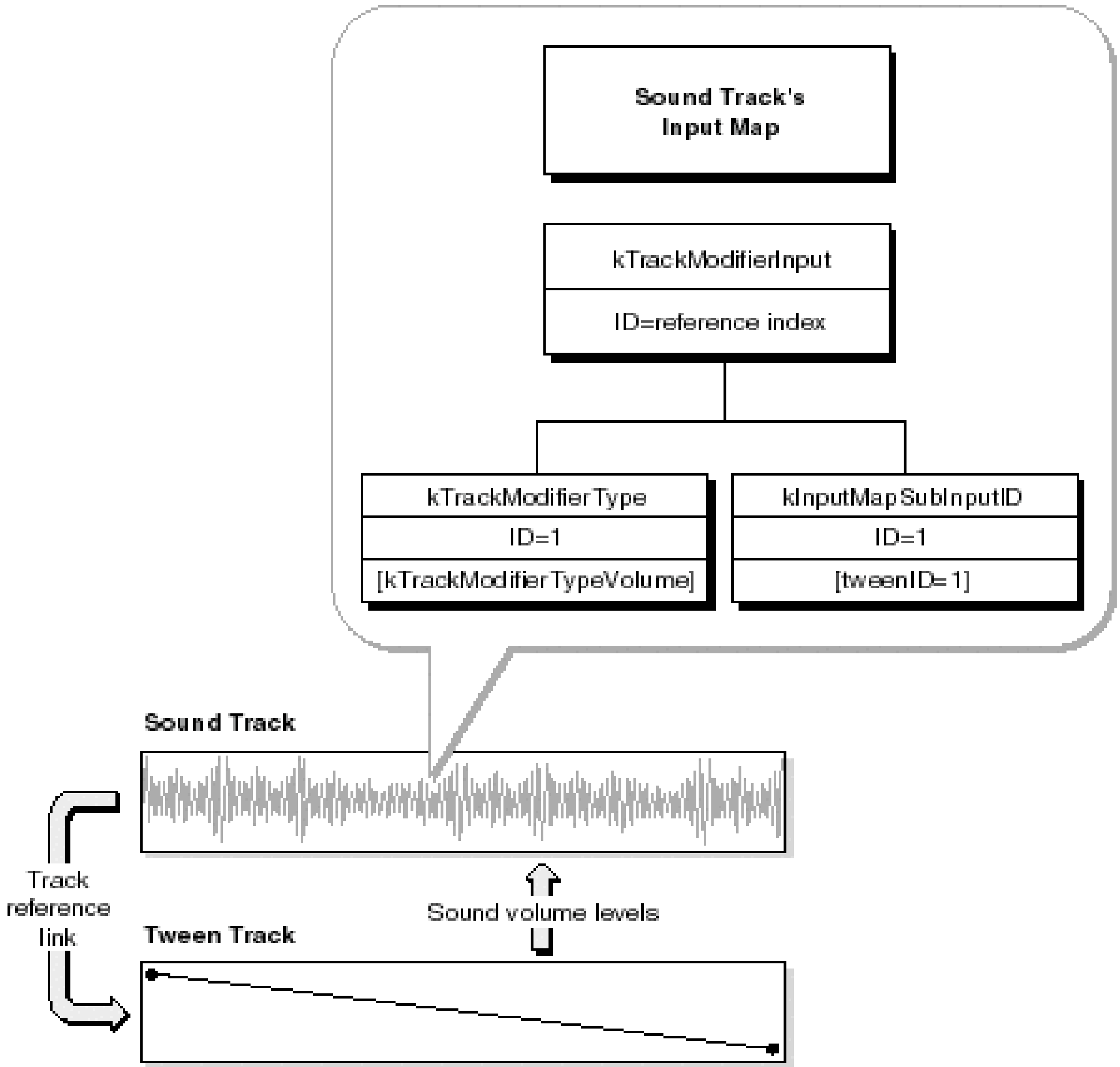
Typical tween components can just interpolate numeric values or can perform complex tweening, such as finding intermediate data between one matrix or polygon and another. These processes are described in Tween Components and Native Tween Types (page 57).

Using the Tween Media Handler

This section describes how to create a tween track and how to connect a tween track to the input map of the track(s) it will modify. Detailed code examples are provided.

You can use the tween media handler to send tween values from a tween track to a receiving track, such as a video track or a sound track. To send tween values, you must create a tween track. The Movie Toolbox routes the data from the tween track to the receiving track based upon the receiving track's input map, as shown in Figure 6-1.

Figure 6-1 Data routed from a tween track to a receiving track based on its input map



Creating a Tween Track

To create a tween track, you must:

1. Create a tween track and its media.
2. Create one or more tween media samples.
3. Add the media samples to the tween media.
4. Add the tween media to the track.

5. Create a link from the tween track to the track to which the tween media handler should send tween values.

6. Bind the tween entry to the desired attributes in the receiving track.

The sample code shown in this section creates a tween sample that interpolates a short integer from 255 to 0. The tween media is attached to the sound track of a QuickTime movie to modify the sound track's volume. Thus it creates a volume fadeout using the tween track. The data type for the tween component is `kTweenTypeShort`.

The sample code shown in Listing 6-1 creates a new track (`t`) to be used as the tween track and new tween media (type `TweenMediaType`).

Listing 6-1 Creating a tween track and tween media

```
Track t;
Media md;
SampleDescriptionHandle desc;
// ...
// set up the movie, m
// ...
// allocate a sample description handle
desc = (SampleDescriptionHandle)NewHandleClear (
    sizeof (SampleDescription));
// create the tween track, t
t = NewMovieTrack (m, 0, 0, kNoVolume);
// create the tween media, md
md = NewTrackMedia (t, TweenMediaType, 600, nil, 0);
(**desc).descSize = sizeof(SampleDescription);
```

Next, your application must create a tween media sample. The tween media sample is a QT atom container structure that contains one or more `kTweenEntry` atoms. Each `kTweenEntry` atom defines a separate tween operation. A single tween sample can describe several parallel tween operations.

The sample code shown in Listing 6-2 creates a new QT atom container and inserts a `kTweenEntry` atom into the container. Then, it creates two leaf atoms, both children of the `kTweenEntry` atom. The first leaf atom (atom type `kTweenType`) contains the type of the tween data, in this case `kTweenTypeShort`. The second leaf atom (atom type `kTweenData`) contains the two data values for the tween operation, 512 and 0.

Remember that all data in QT atoms must be big-endian. The sample code shown in this section contains the endian conversion routines required for cross-platform compatibility.

Listing 6-2 Creating a tween sample

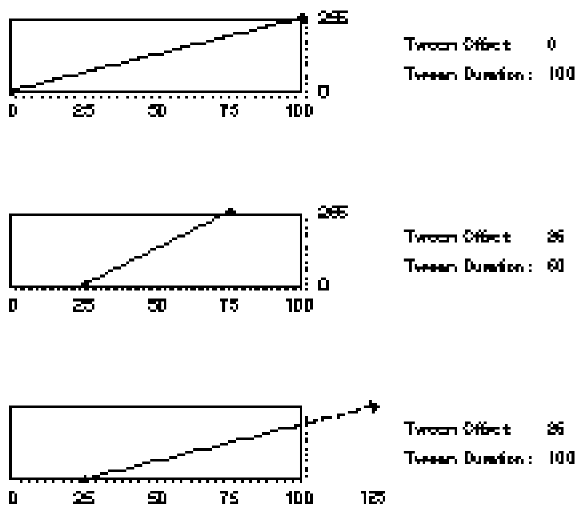
```
QTAtomContainer container = nil;
short tweenDataShort[2];
QTAtomType tweenType;
tweenDataShort[0] = EndianS16_NtoB(255);
tweenDataShort[1] = EndianS16_NtoB(0);
// create a new atom container to hold the sample
QTNewAtomContainer (&container);
// create the parent tween entry atom
QTInsertChild (container, kParentAtomIsContainer, kTweenEntry, 1, 0, 0,
    nil, &tweenAtom);
// add two child atoms to the tween entry atom
```

```
// * the type atom, kTweenType
tweenType = EndianU32_NtoB(kTweenTypeShort);
QTInsertChild (container, tweenAtom, kTweenType, 1, 0,
    sizeof(tweenType), &tweenType, nil);
// * the data atom, kTweenData
QTInsertChild (container, tweenAtom, kTweenData, 1, 0, sizeof(short) * 2,
    tweenDataShort, nil);
```

You do not have to start the tween at the beginning of the sample, nor do you have to stop at the end of the sample. You can specify the start of the tween and its duration by adding additional child atoms to the tween entry.

Figure 6-2 illustrates how you can use tween media to modify a sound track's volume. The first part of the illustration shows an example of tweening the sound volume from 0 to 255, with the tween offset at 0 and the tween duration at 100. In the second part, with the tween offset at 25 and the duration at 50, the tween has no effect until time 25, after which it causes the volume to fade in over the next 50 time units. The volume is left at 255. The third part shows the tween offset at 25 and the tween duration at 100. Since the offset plus the duration of this tween is greater than the duration of the tween media sample, the sound track never reaches full volume.

Figure 6-2 Using tween media to modify the sound track's volume



You can add a `kTweenStartOffset` atom to start the tween operation at 500 units into the sample with the following lines of code:

```
TimeValue time = EndianU32_NtoB(500);
QTInsertChild (container, tweenAtom, kTweenStartOffset, 1, 0,
    sizeof(TimeValue), &time, nil);
```

You can specify a duration for the tween operation independent of the sample's duration by adding a `kTweenDuration` atom to the tween entry, as follows:

```
TimeValue duration = EndianU32_NtoB(1000);
QTInsertChild (container, tweenAtom, kTweenDuration, 1, 0,
    sizeof(TimeValue), &duration, nil);
```

Once the tween samples have been created, you can add them to the tween media and then add the tween media to the track, as shown in Listing 6-3.

Listing 6-3 Adding the tween sample to the media and the media to the track

```
// add the sample to the tween media
BeginMediaEdits (md);
AddMediaSample (md, container, 0,
    GetHandleSize(container), kSampleDuration, desc, 1, 0, nil);
EndMediaEdits(md);
// dispose of the sample description handle and the atom container
DisposeHandle ((Handle)desc);
QTDisposeAtomContainer(container);
// add the media to the track
InsertMediaIntoTrack(t, 0, 0, kSampleDuration, kFix1);
```

Once you have added the tween media to its track, you need to call the `AddTrackReference` function to create a link between the tween track to the receiving track. `AddTrackReference` returns the index of the reference it creates.

The sample code shown in Listing 6-4 retrieves the sound track from a movie and calls `AddTrackReference` to create a link between the tween track (`t`) and the sound track. The reference index is returned in the parameter `referenceIndex`.

Listing 6-4 Creating a link between the tween track and the sound track

```
Track soundTrack;
long referenceIndex;
// retrieve the sound track from the movie
soundTrack = GetMovieIndTrackType (theMovie, 1,
    AudioMediaCharacteristic,
    movieTrackCharacteristic | movieTrackEnabledOnly);
// create a link between the tween track and the sound track --
// on return, referenceIndex contains the index of the link
err = AddTrackReference (soundTrack, t, kTrackModifierReference,
                            &referenceIndex);
```

Once you have linked the tween track to its receiving track, you must update the input map of the receiving track's media to indicate how the receiving track should interpret the data it receives from the tween track.

To do this, you create a QT atom container and insert an atom of type `kTrackModifierInput` whose ID is the index returned by the `AddTrackReference` function. Then, you insert two atoms as children of the `kTrackModifierInput` atom:

- A leaf atom of type `kTrackModifierType` that contains the attribute of the receiving track to be modified. For example, if the tween entry modifies the matrix of the track, the leaf atom would contain the type `kTrackModifierTypeMatrix`.
- A leaf atom of type `kInputMapSubInputID` that contains the ID of the tween entry atom. This binds the tween entry to the receiving track.

Once you have created the appropriate atoms in the input map, you call `SetMediaInputMap` to assign the input map to the receiving track's media.

The code shown in Listing 6-5 creates an input map for the sound track of a movie. In this code, the tween media is linked to a sound track; the interpolated tween values are used to modify the sound track's volume.

Listing 6-5 Binding a tween entry to its receiving track

```
QTAtomContainer inputMap = nil;
// create an atom container to hold the input map
if (QTNewAtomContainer (&inputMap) == noErr)
{
    QTAtom inputAtom;
    OSType inputType;
    long tweenID = 1;
    // create a kTrackModifierInput atom
    // whose ID is referenceIndex
    QTInsertChild(inputMap, kParentAtomIsContainer,
        kTrackModifierInput, referenceIndex, 0, 0, nil,
        &inputAtom);
    // add a child atom of type kTrackModifierTypeVolume
    inputType = EndianU32_NtoB(kTrackModifierTypeVolume);
    QTInsertChild (inputMap, inputAtom, kTrackModifierType, 1, 0,
      sizeof(inputType), &inputType, nil);
    // add a child atom for the ID of the tween to
    // modify the volume
    QTInsertChild (inputMap, inputAtom, kInputMapSubInputID, 1,
        0, sizeof(tweenID), &tweenID, nil);
    // assign the input map to the sound media
    SetMediaInputMap(GetTrackMedia(soundTrack), inputMap);
    // dispose of the input map
    QTDisposeAtomContainer(inputMap);
}
```

Tween Media Handler Constants

This section defines a QT atom type used for mapping a tween to the receiving track of a movie.

The following input type is defined for tween-related atoms:

```
enum {
    kInputMapSubInputID         = 'subi',
};
```

The `kInputMapSubInputID` type is the QT atom type for mapping a tween to a receiving track in a movie:

Term	Definition
`kInputMapSubInputID`	A leaf atom that contains the ID of a tween entry. You create a `kInputMapSubInputID` atom in a receiving track's input map to define the relationship between the tween entry and the receiving track.You create a `kInputMapSubInputID` atom as a child of a `kTrackModifierInput` atom.

Using Tween Components

This chapter describes how to create tween containers that the tween media handler uses. Several utility routines are also discussed.

The tween media handler uses tween components to perform tween operations (other than simple ones built into QuickTime). The components use containers to define their tween operations, and the containers are constructed from QT atoms. The tween media handler is discussed in Tween Media Handlers (page 29).

Creating a Single Tween Container

To create a single tween container, do the following:

1. Create a QT atom container.
2. Insert a `kTweenEntry` atom into the QT atom container for the tween. A `kTweenEntry` atom contains the atoms that define the tween.
3. Insert a `kTweenType` atom that specifies the tween type into the `kTweenEntry` atom.
4. Insert a `kTweenData` atom that contains the data for the tween into the `kTweenEntry` atom.

Listing 7-1 shows how to create a single `kTweenTypeLong` tween container that a component could use to interpolate two long integers.

Listing 7-1 Creating a single kTweenTypeLong tween container

```
QTAtomContainer container = nil;
long tweenDataLong[2];
QTAtomType tweenType;
QTAtom tweenAtom;
tweenDataLong[0] = EndianU32_NtoB(512);
tweenDataLong[1] = EndianU32_NtoB(0);
// create a new atom container
QTNewAtomContainer (&container);
// create the parent tween entry atom
tweenType = kTweenTypeLong;
QTInsertChild (container, kParentAtomIsContainer, kTweenEntry, 1, 0, 0,
                nil, &tweenAtom);
// add two child atoms to the tween entry atom --
// * the type atom, kTweenType
QTInsertChild (container, tweenAtom, kTweenType, 1, 0,
                sizeof(tweenType), &tweenType, nil);
// * the data atom, kTweenData
QTInsertChild (container, tweenAtom, kTweenData, 1, 0,
                sizeof(long) * 2, tweenDataLong, nil);
```

Using Path Tween Components

The following sections describe how to use a variety of path tween components. All path tween operations have as input a QuickTime vector data stream for a path. Path tweeners interpret their time input in one of these ways:

- As a percentage of path's length. For these types, either a point on the path or a `MatrixRecord` data structure is returned.
- As a function. One operation returns the `y` value of the point on the path with a given `x` value, and the other returns the `x` value of the point on the path with a given `y` value.

If the `kTweenReturnDelta` flag (in an optional `kTweenFlags` atom in the `kTweenEntry` atom) is set, a path tween returns the change in value from the last time it was invoked. If the flag is not set, or if the component has not previously been invoked, the component returns the normal result for the tween.

Using a List Tween Component

To use a list tween component (of type `kTweenTypeAtomList`), do the following:

1. Create a QT atom container.
2. Insert a `kTweenEntry` atom into the QT atom container for the tween.
3. Insert a `kTweenType` atom that specifies the tween type into the `kTweenEntry` atom.
4. Insert a `kTweenData` atom into the `kTweenEntry` atom. Unlike the previous example, this `kTweenData` atom is not a leaf atom.
5. Insert a `kListElementType` atom that specifies the atom type of the list entries into the `kTweenData` atom. The list entries must be leaf atoms.
6. Insert leaf atoms of the type specified by the `kListElementType` atom into the `kTweenData` atom.

The duration of the tween operation is divided by the number of leaf atoms of the specified type. For time points within the first time division (from the start of the duration up to an including the time (*total time / number of atoms*)), the data for the first leaf atom is returned; for the second time division, the data for the second leaf atom is returned; and so on.

Listing 7-2 shows how to create a list tween.

Listing 7-2 Creating a kTweenTypeAtomList tween container

```
QTAtomContainer container = nil;
long tweenDataLong[2];
QTAtomType tweenType;
QTAtom tweenAtom;
tweenDataLong[0] = EndianU32_NtoB(512);
tweenDataLong[1] = EndianU32_NtoB(0);
// create a new atom container to hold the sample
```

```
QTNewAtomContainer (&container);
// create the parent tween entry atom
tweenType = EndianU32_NtoB(kTweenTypeLong);
QTInsertChild (container, kParentAtomIsContainer, kTweenEntry, 1, 0, 0,
                nil, &tweenAtom);
// add two child atoms to the tween entry atom --
// * the type atom, kTweenType
QTInsertChild (container, tweenAtom, kTweenType, 1, 0,
                sizeof(tweenType), &tweenType, nil);
// * the data atom, kTweenData
QTInsertChild (container, tweenAtom, kTweenData, 1, 0,
                sizeof(short) * 2, tweenDataLong, nil);

OSErr       err = noErr;
QTTweener tween;
QTAtomContainer container = nil, listContainer = nil;
OSType      tweenerType;
TimeValue   offset, duration, tweenTime;
Handle      result;
QTAtom      tweenAtom;
tweenerType = EndianU32_NtoB(kTweenTypeAtomList);
offset  = 0;
duration = 3;
err = QTNewAtomContainer( &container );
if ( err ) goto bail;
err = AddTweenAtom( container, kParentAtomIsContainer, 1, tweenerType,
                    offset, duration, 0, 0, nil, &tweenAtom );
if ( err ) goto bail;
listContainer = CreateSampleAtomListTweenData( 1 );
if ( listContainer == nil ) { err = memFullErr; goto bail; }
err = AddDataAtom( container, tweenAtom, 1, 0, nil,
                    listContainer, 0, nil );
if ( err ) goto bail;
err = QTNewTween( &tween, container, tweenAtom, duration );
if ( err ) goto bail;
result = NewHandle( 0 );
if ( err = MemError() ) goto bail;
    // exercise the AtomListTweener
    for ( tweenTime = 1; tweenTime <= duration; tweenTime += 1 ) {
        long pictureID;

        err = QTDoTween( tween, tweenTime, result, nil, nil, nil );
        if ( err ) goto bail;

        // the pictureID from the atomDataList corresponding to tweenTime
        pictureID = *(long *)*result;
    }

    err = QTDisposeTween( tween );
bail:
    if ( container ) QTDisposeAtomContainer( container );
    if ( listContainer ) QTDisposeAtomContainer( listContainer );
    if ( result ) DisposeHandle( result );
    return err;
}
```

Utility Routines

The examples in the next sections use several utility routines to modularize their code. These routines are the following:

- AddTweenAtom adds an atom of type kTweenEntry plus its standard child atoms (other than the kTweenDataAtom) to a container.
- AddDataAtom adds a data atom with an ID of dataAtomID as a child atom of tweenAtom.
- AddSequenceTweenAtom adds a kTweenEntry atom for a sequenced tween.
- AddSequenceElement adds a leaf atom of type kTweenSequenceElement to a container.

AddTweenAtom

Listing 7-3 shows AddTweenAtom, a routine that adds an atom of type kTweenEntry plus its standard child atoms (other than the kTweenDataAtom) to a container. Only the tweenAtomID and tweenerType parameters are required; you may pass 0 for the other parameters to avoid using them.

The minOutput and maxOutput atom parameters are necessary only if the tweener is being used as an interpolator. Passing a tweenAtomID value of 0 means that the routine can assign any unique ID.

If you use AddTweenAtom to add a nonsequenced tween entry to a container, the kTweenEntry atom it creates is the atom you pass to QTNewTween and the sequenceAtom you pass in may be kParentAtomIsContainerTween Components and Tween Media. In this case the tweenAtomID value should be 1.

If you use AddTweenAtom to add a sequenced tween entry to a container, the newSequenceAtom returned by AddSequenceTweenAtom is its sequenceAtom parameter and will also be the atom you pass to QTNewTween. Note that in most cases a sequenced tween contains only one tween entry but may contain multiple data atoms.

All tween atoms within same sequenceAtom must have same tween type. It is unlikely that you would want to change the duration or offset values within the same sequenceAtom.

Listing 7-3 Utility routine AddTweenAtom

```
OSErr AddTweenAtom( QTAtomContainer container, QTAtom sequenceAtom,
    QTAtomID tweenAtomID, OSType tweenerType, TimeValue offset,
    TimeValue duration, Fixed minOutput, Fixed maxOutput, StringPtr name,
    QTAtom *newTweenAtom )
{
    OSErr       err = noErr;
    QTAtom      tweenAtom = 0;
    if ( ! container )      { err = paramErr; goto bail; }
    err = QTInsertChild( container, sequenceAtom, kTweenEntry,
                        tweenAtomID, 0, 0, nil, &tweenAtom );
    if ( err ) goto bail;
    err = QTInsertChild( container, tweenAtom, kTweenType, 1, 1,
                        sizeof(tweenerType), &tweenerType, nil );
    if ( err ) goto bail;
    if ( offset ) {
```

```
        err = QTInsertChild( container, tweenAtom, kTweenStartOffset, 1,
                                1, sizeof(offset), &offset, nil );
        if ( err ) goto bail;
    }
    if ( duration ) {
        err = QTInsertChild( container, tweenAtom, kTweenDuration, 1, 1,
                                sizeof(duration), &duration, nil );
        if ( err ) goto bail;
    }
    // default minOutput is zero, so this is OK
    if ( minOutput ) {
        err = QTInsertChild( container, tweenAtom, kTweenOutputMin, 1, 1,
                                sizeof(minOutput), &minOutput, nil );
        if ( err ) goto bail;
    }

    if ( maxOutput ) {
        err = QTInsertChild( container, tweenAtom, kTweenOutputMax, 1, 1,
                                sizeof(maxOutput), &maxOutput, nil );
        if ( err ) goto bail;
    }
    if ( name ) {
        err = QTInsertChild( container, tweenAtom, kNameAtom, 1, 1,
                                name[0] + 1, name, nil );
        if ( err ) goto bail;
    }
bail:
    if ( newTweenAtom )
        *newTweenAtom = tweenAtom;
    return err;
}
```

AddDataAtom

Listing 7-4 shows `AddDataAtom`, a routine that adds a data atom with an ID of `dataAtomID` as a child atom of `tweenAtom`. If `dataSize` is nonzero, then leaf data is copied from `dataPtr` to `dataAtom`. Otherwise (if `dataContainer` in not `nil`Tween Components and Tween Media), child atoms are copied from `dataContainer`. If you wish to add the actual data by using another routine, you may pass 0 in `dataSize` and `nil` in both `dataPtr` and `dataContainer`.

You can associate a tweener to be used as an interpolator for each `dataAtom` value. The `interpolationTweenID` parameter specifies the ID of a `kTweenEntry` atom that is a child of the `newSequenceAtom` returned by `AddSequenceTweenAtom`. If you specify an interpolation tweener, then the `atTime` parameter of the `DoTween` routine is first fed as an input to the interpolation tweener. The `tweenResult` of the interpolation tweener becomes the `atTime` parameter of the succeeding tweener. Note that the `kTweenData` atom and the `kTweenInterpolationID` atom have the same ID; this is how QuickTime groups them together.

For best performance, the output range of an interpolation tweener should be from 0 to the duration of the regular tweener. However, you may specify the minimum and maximum values that the interpolation tweener returns; this lets the tweener be shared. The minimum and maximum values are used to scale `tweenResult`, and are added as child atoms of a `kTweenEntry` in `AddTweenAtom`.

Listing 7-4 Utility routine AddDataAtom

```
OSErr AddDataAtom( QTAtomContainer container, QTAtom tweenAtom,
    QTAtomID dataAtomID, long dataSize, Ptr dataPtr,
    QTAtomContainer dataContainer, QTAtomID interpolationTweenID,
    QTAtom *newDataAtom )
{
    OSErr       err = noErr;
    QTAtom      dataAtom = 0;
    if ( (! container) || (dataAtomID == 0) || (dataSize &&
        (dataContainer || !dataPtr)) )  { err = paramErr; goto bail; }
    err = QTInsertChild( container, tweenAtom, kTweenData, dataAtomID, 0,
                            dataSize, dataPtr, &dataAtom );
    if ( err ) goto bail;
    if ( dataSize ) {
        err = QTSetAtomData( container, dataAtom, dataSize, dataPtr );
        if ( err ) goto bail;
    }
    else if ( dataContainer ) {
        err = QTInsertChildren( container, dataAtom, dataContainer );
        if ( err ) goto bail;
    }
    if ( interpolationTweenID ) {
        err = QTInsertChild( container, tweenAtom, kTweenInterpolationID,
                            dataAtomID, 0, sizeof(interpolationTweenID),
                            &interpolationTweenID, nil );
        if ( err ) goto bail;
    }
bail:
    if ( newDataAtom )
        *newDataAtom = dataAtom;
    return err;
}
```

AddSequenceTweenAtom

Listing 7-5 shows `AddSequenceTweenAtom`, a routine that adds a `kTweenEntry` atom for a sequenced tween. The `newSequenceAtom` returned may be passed into the `AddTweenAtom` and `AddSequenceElement` routines as their `sequenceAtom` parameter.

To create a nonsequenced tween, use `AddTweenAtom` instead.

Listing 7-5 Utility routine AddSequenceTweenAtom

```
OSErr AddSequenceTweenAtom( QTAtomContainer container, QTAtom parentAtom,
                    QTAtomID sequenceAtomID, QTAtom *newSequenceAtom )
{
    if ( (! container) || (sequenceAtomID == 0) )  { return paramErr; }
    return QTInsertChild( container, parentAtom, kTweenEntry,
                        sequenceAtomID, 0, 0, nil, newSequenceAtom );
}
```

AddSequenceElement

Listing 7-6 shows `AddSequenceElement`, a routine that adds a leaf atom of type `kTweenSequenceElement` to a container. The `sequenceAtom` that you pass in is the `newSequenceAtom` parameter returned by `AddSequenceTweenAtom`. Each time you call `AddSequenceElement`, a sequence tween element is added to the end of the list of elements. The tween toolbox organizes the list in index order, with IDs ignored.

Each element has a duration that is some percentage of the tween's duration. The element's duration is its `endPercent` value minus the previous element's `endPercent` value. For example, if you wanted three elements to last 0.25, 0.25, and 0.5 of the tween's duration, then the elements' `endPercent` values should be set to 0.25, 0.5, and 1. The elements tell the tween toolbox which `tweenAtom` and `dataAtom` to switch to.

The `tweenAtomID` is the ID of a `kTweenEntry` atom within the `sequenceAtom`. The `dataAtomID` is the ID of a `kTweenData` atom. The `kTweenData` atom is a child atom of the specified `tweenAtom`. Usually you only need to create one `tweenAtom` with multiple data atoms. Some tweener types (such as 3D tweeners) use child atoms of the `tweenEntry` atom for initialization, so in these cases you usually create a `tweenAtom` with one `dataAtom` per sequence entry.

Listing 7-6 Utility routine AddSequenceElement

```
OSErr AddSequenceElement( QTAtomContainer container, QTAtom sequenceAtom,
    Fixed endPercent, QTAtomID tweenAtomID, QTAtomID dataAtomID,
    QTAtom *newSequenceElementAtom )
{
    TweenSequenceEntryRecord    entry;
    if ( (! container) || (endPercent > (1L<<16)) || (tweenAtomID == 0)
                            || (dataAtomID == 0) ){ return paramErr; }
    entry.endPercent        = endPercent;
    entry.tweenAtomID       = tweenAtomID;
    entry.dataAtomID        = dataAtomID;
    // adds at end of list by index, with any unique atom id
    return QTInsertChild( container, sequenceAtom, kTweenSequenceElement,
                0, 0, sizeof(entry), &entry, newSequenceElementAtom );
}
```

CreateSampleAtomListTweenData

Listing 7-7 shows the `CreateSampleAtomListTweenData` routine.

Listing 7-7 Utility routine CreateSampleAtomListTweenData

```
QTAtomContainer CreateSampleAtomListTweenData( long whichOne )
{
    OSErr               err = noErr;
    QTAtomContainer     atomListContainer;
    QTAtomType          tweenAtomType;
    UInt32              elementDataType;
    UInt16              resourceID;

    err = QTNewAtomContainer( &atomListContainer );
    if ( err ) goto bail;
```

```
    // kListElementDataType atom specifies the data type of the elements
    elementDataType = kTweenTypeShort;
    err = QTInsertChild( atomListContainer, kParentAtomIsContainer,
                        kListElementDataType, 1, 1,
                        sizeof(tweenAtomType),
                        &tweenAtomType, nil );
    // kListElementType atom tells which type of atoms to look for
    tweenAtomType = 'pcid';
    err = QTInsertChild( atomListContainer, kParentAtomIsContainer,
                        kListElementType, 1, 1, sizeof(tweenAtomType),
                        &tweenAtomType, nil );
    switch ( whichOne ) {
        case 1:
            resourceID = 1000;
            err = QTInsertChild( atomListContainer,
                        kParentAtomIsContainer, 'pcid', 1, 1,
                        sizeof(resourceID), &resourceID, nil );
            if ( err ) goto bail;

            resourceID = 1001;
            err = QTInsertChild( atomListContainer,
                        kParentAtomIsContainer, 'pcid', 2, 2,
                        sizeof(resourceID), &resourceID, nil );
            if ( err ) goto bail;
            resourceID = 1002;
            err = QTInsertChild( atomListContainer,
                        kParentAtomIsContainer, 'pcid', 3, 3,
                        sizeof(resourceID), &resourceID, nil );
            if ( err ) goto bail;
        break;

        case 2:
            resourceID = 1003;
            err = QTInsertChild( atomListContainer,
                        kParentAtomIsContainer, 'pcid', 1, 1,
                        sizeof(resourceID), &resourceID, nil );
            if ( err ) goto bail;

            resourceID = 1004;
            err = QTInsertChild( atomListContainer,
                        kParentAtomIsContainer, 'pcid', 2, 2,
                        sizeof(resourceID), &resourceID, nil );
            if ( err ) goto bail;
            resourceID = 1005;
            err = QTInsertChild( atomListContainer,
                        kParentAtomIsContainer, 'pcid', 3, 3,
                        sizeof(resourceID), &resourceID, nil );
            if ( err ) goto bail;
        break;
    }

bail:
    if ( err && atomListContainer )
        { QTDisposeAtomContainer( atomListContainer );
          atomListContainer = nil; }
    return atomListContainer;
}
```

Using a Polygon Tween Component

A polygon tweener maps a four-sided polygon, such as the boundary of a sprite or track, into another. It can be used to create perspective effects, in which the shape of the destination polygon changes over time. The range of polygons into which the source polygon is mapped is defined by two additional four-sided polygons, which are interpolated to specify a destination polygon for any time point in the tween duration.

To use a polygon tween component (of type `kTweenTypePolygonTween` Components and Tween Media), do the following:

1. Create a QT atom container.
2. Insert a `kTweenEntry` atom into the QT atom container for the tween.
3. Insert a `kTweenType` atom that specifies the tween type into the `kTweenEntry` atom.
4. Insert a `kTweenData` atom into the `kTweenEntry` atom.

The data is an array of 27 fixed-point values (`Fixed[27]Tween` Components and Tween Media) that specifies the three four-sided polygons. Each polygon is specified by 9 consecutive array elements. The first element is each set of 9 contains the number of points used to specify the polygon; this value is coerced to a long integer, and it must always be 4 after coercion. The following 8 values in each set of nine are four `x`, `y` pairs that specify the corners of the polygon.

The first set of 9 elements specifies the dimensions of a sprite or track to be mapped. For example, if the object is a sprite, the four points are (0,0), (*spriteWidth, 0*), (*spriteWidth, spriteHeight),* (0, *spriteHeight).* The next set of 9 elements specifies the initial polygon into which the sprite or track is mapped. The next set of 9 elements specifies the final polygon into which the sprite or track is mapped.

The output is a `MatrixRecord` data structure that you use to map the sprite or track into a four-sided polygon.

Listing 7-8 shows how to create a polygon tween.

Listing 7-8 Creating a polygon tween container

```
OSErr CreateSamplePolygonTweenContainer( QTAtomContainer container,
    TimeValue duration, QTAtom *newTweenAtom )
{
    OSErr           err = noErr;
    TimeValue       offset;
    Handle          thePolygonData = nil;
    QTAtom          tweenAtom;

    err = QTRemoveChildren( container, kParentAtomIsContainer );
    if ( err ) goto bail;
    offset = 0;
    err = AddTweenAtom( container, kParentAtomIsContainer, 1,
                        kTweenTypePolygon, offset, duration, 0, 0,
                        nil, &tweenAtom );
    if ( err ) goto bail;

    thePolygonData = CreateSamplePolygonData();
    if ( thePolygonData == nil ) { err = memFullErr; goto bail; }
```

```
    HLock( thePolygonData );

    err = AddDataAtom( container, tweenAtom, 1,
                        GetHandleSize( thePolygonData ),
                        *thePolygonData, nil, 0, nil );
    if ( err ) goto bail;
bail:
    if ( thePolygonData ) DisposeHandle( thePolygonData );
    if ( newTweenAtom ) *newTweenAtom = tweenAtom;
    return err;
}

Handle CreateSamplePolygonData( void )
{
    OSErr       err = noErr;
    Handle      polygonData;
    Fixed       *poly;
    polygonData = NewHandle( 27 * sizeof(Fixed) );
    if ( polygonData == nil ) { err = memFullErr; goto bail; }

    poly = (Fixed *)*polygonData;

    poly[0] = EndianU32_NtoB(4);                // source dimensions
    poly[1] = EndianU32_NtoB(Long2Fix( 0 ));
    poly[2] = EndianU32_NtoB(Long2Fix( 0 ));
    poly[3] = EndianU32_NtoB(Long2Fix( 100 ));
    poly[4] = EndianU32_NtoB(Long2Fix( 0 ));
    poly[5] = EndianU32_NtoB(Long2Fix( 100 ));
    poly[6] = EndianU32_NtoB(Long2Fix( 100 ));
    poly[7] = EndianU32_NtoB(Long2Fix( 0 ));
    poly[8] = EndianU32_NtoB(Long2Fix( 100 ));
    poly[9] = EndianU32_NtoB(4);                // tween from polygon
    poly[10] = EndianU32_NtoB(Long2Fix( 100 ));
    poly[11] = EndianU32_NtoB(Long2Fix( 100 ));
    poly[12] = EndianU32_NtoB(Long2Fix( 200 ));
    poly[13] = EndianU32_NtoB(Long2Fix( 100 ));
    poly[14] = EndianU32_NtoB(Long2Fix( 200 ));
    poly[15] = EndianU32_NtoB(Long2Fix( 200 ));
    poly[16] = EndianU32_NtoB(Long2Fix( 100 ));
    poly[17] = EndianU32_NtoB(Long2Fix( 200 ));
    poly[18] = EndianU32_NtoB(4);               // tween to polygon
    poly[19] = EndianU32_NtoB(Long2Fix( 140 ));
    poly[20] = EndianU32_NtoB(Long2Fix( 100 ));
    poly[21] = EndianU32_NtoB(Long2Fix( 160 ));
    poly[22] = EndianU32_NtoB(Long2Fix( 100 ));
    poly[23] = EndianU32_NtoB(Long2Fix( 200 ));
    poly[24] = EndianU32_NtoB(Long2Fix( 200 ));
    poly[25] = EndianU32_NtoB(Long2Fix( 100 ));
    poly[26] = EndianU32_NtoB(Long2Fix( 200 ));
bail:
    return polygonData;
}
```

Specifying an Offset for a Tween Operation

You can start a tween operation after a tween media sample begins by including an optional `kTweenStartOffset` atom in the `kTweenEntry` atom for the tween. This atom specifies a time interval, beginning at the start of the tween media sample, after which the tween operation begins. If this atom is not included, the tween operation begins at the start of the tween media sample.

Specifying a Duration for a Tween

You can specify the duration of a tween operation by including an optional `kTweenDuration` atom in the `kTweenEntry` atom for the tween. When a QuickTime movie includes a tween track, the time units for the duration are those of the tween track's time scale. If a tween component is used outside of a movie, the application using the tween data determines how the duration value and values returned by the component are interpreted.

Creating a Tween Sequence

Single Tweens and Tween Sequences (page 26) discussed tween sequences, in which different tween operations of the same type may be applied sequentially. The type `kTweenSequenceElement` specifies an entry in a tween sequence. Its parent is the tween QT atom container (which you specify with the constant `kParentAtomIsContainerTween` Components and Tween Media).

The ID of a `kTweenSequenceElement` atom must be unique among the `kTweenSequenceElement` atoms in the same QT atom container. The index of a `kTweenSequenceElement` atom specifies its order in the sequence; the first entry in the sequence has the index 1, the second 2, and so on.

This atom is a leaf atom. The data type of its data is `TweenSequenceEntryRecord`, a data structure that contains the following fields:

- `endPercent`, a value of type `Fixed` that specifies the point in the duration of the tween media sample at which the sequence entry ends. This is expressed as a fraction; for example, if the value is 0.75, the sequence entry ends after three-quarters of the total duration of the tween media sample has elapsed. The sequence entry begins after the end of the previous sequence entry or, for the first entry in the sequence, at the beginning of the tween media sample.
- `tweenAtomID`, a value of type `QTAtomID` that specifies the `kTweenEntry` atom containing the tween for the sequence element. The `kTweenEntry` atom and the `kTweenSequenceElement` atom must both be child atoms of the same tween QT atom container.
- `dataAtomID`, a value of type `QTAtomID` that specifies the `kTweenData` atom containing the data for the tween. This atom must be a child atom of the atom specified by the `tweenAtomID` field.

Listing 7-9 shows how to create a tween sequence.

Listing 7-9 Creating a tween sequence

```
OSErr CreateSampleSequencedTweenContainer( QTAtomContainer container,
    TimeValue duration, QTAtom *newTweenAtom )
```

```
{
    OSErr               err = noErr;
    QTAtomContainer     dataContainer = nil;
    OSType              tweenerType;
    QTAtom              sequenceAtom, tweenAtom;
    TimeValue           offset;
    Handle              result;
    QTAtomID            tweenAtomID, dataAtomID;
    Fixed               endPercent;

    err = QTRemoveChildren( container, kParentAtomIsContainer );
    if ( err ) goto bail;
    tweenerType = kTweenTypeAtomList;
    offset  = 0;
    err = AddSequenceTweenAtom( container, kParentAtomIsContainer,
                                1, &sequenceAtom );
    if ( err ) goto bail;

    offset = 0;

    err = AddTweenAtom( container, sequenceAtom, 1, tweenerType, offset,
                        duration, 0, 0, nil, &tweenAtom );
    if ( err ) goto bail;

    // add first data atom (id 1) to tween atom
    dataAtomID = 1;
    dataContainer = CreateSampleAtomListTweenData( dataAtomID );
    if ( ! dataContainer ) { err = memFullErr; goto bail; }
    err = AddDataAtom( container, tweenAtom, dataAtomID, 0, nil,
                        dataContainer, 0, nil );
    if ( err ) goto bail;

    QTDisposeAtomContainer( dataContainer );

    // add second data atom (id 2) to tween atom
    dataAtomID = 2;
    dataContainer = CreateSampleAtomListTweenData( dataAtomID );
    if ( ! dataContainer ) { err = memFullErr; goto bail; }
    err = AddDataAtom( container, tweenAtom, dataAtomID, 0, nil,
                        dataContainer, 0, nil );
    if ( err ) goto bail;

    QTDisposeAtomContainer( dataContainer );
    // now create a sequence with four elements; the first three are data
    // atom 1, the last is data atom 2
    endPercent = FixDiv( Long2Fix(25), Long2Fix(100) );
    tweenAtomID = 1;
    dataAtomID = 1;
    err = AddSequenceElement( container, sequenceAtom, endPercent,
                                tweenAtomID, dataAtomID, nil );
    if ( err ) goto bail;
    endPercent = FixDiv( Long2Fix(50), Long2Fix(100) );
    tweenAtomID = 1;
    dataAtomID = 1;
    err = AddSequenceElement( container, sequenceAtom, endPercent,
                                tweenAtomID, dataAtomID, nil );
    if ( err ) goto bail;
    endPercent = FixDiv( Long2Fix(75), Long2Fix(100) );
```

```
    tweenAtomID = 1;
    dataAtomID = 1;
    err = AddSequenceElement( container, sequenceAtom, endPercent,
                              tweenAtomID, dataAtomID, nil );
    if ( err ) goto bail;
    endPercent = FixDiv( Long2Fix(100), Long2Fix(100) );
    tweenAtomID = 1;
    dataAtomID = 2;
    err = AddSequenceElement( container, sequenceAtom, endPercent,
                              tweenAtomID, dataAtomID, nil );
    if ( err ) goto bail;
bail:
    if ( err ) {
        if ( container )
            QTRemoveChildren( container, kParentAtomIsContainer );
        *newTweenAtom = nil;
    }
    else
        *newTweenAtom = sequenceAtom;
}
```

Naming Tweens

You can use the `kNameAtom` atom to store a string value containing a name (or any other information) in a tween container. This atom is not required and is not routinely accessed by QuickTime. It is available for use by your authoring tools or other software.

CreateSampleVectorData Utility

Listing 7-10 shows the `CreateSampleVectorData` routine.

Listing 7-10 Utility routine CreateSampleVectorData

```
Handle CreateSampleVectorData( long whichOne )
{
    OSErr               err;
    Handle              pathData = nil, vectorData = nil;
    ComponentInstance   ci = nil;
    gxPoint             aPoint;

    err = OpenADefaultComponent( decompressorComponentType,
                                 kVectorCodecType, &ci );
    if ( err ) goto bail;
    err = CurveNewPath( ci, &pathData );
    if ( err ) goto bail;
    if ( pathData == nil )
        { err = memFullErr; goto bail; }
    switch ( whichOne ) {
        case 1:
            aPoint.x = Long2Fix( 0 );
            aPoint.y = Long2Fix( 100 );
```

```
            err = CurveInsertPointIntoPath( ci, &aPoint, pathData,
                                            0, 0, false );
            if ( err ) goto bail;

            aPoint.x = Long2Fix( 100 );
            aPoint.y = Long2Fix( 0 );
            err = CurveInsertPointIntoPath( ci, &aPoint, pathData,
                                            0, 1, false );
            if ( err ) goto bail;
            aPoint.x = Long2Fix( 200 );
            aPoint.y = Long2Fix( 100 );
            err = CurveInsertPointIntoPath( ci, &aPoint, pathData,
                                            0, 2, false );
            if ( err ) goto bail;
            aPoint.x = Long2Fix( 100 );
            aPoint.y = Long2Fix( 200 );
            err = CurveInsertPointIntoPath( ci, &aPoint, pathData,
                                            0, 3, false );
            if ( err ) goto bail;
        break;
        case 2:
            aPoint.x = 0;
            aPoint.y = 100;
            err = CurveInsertPointIntoPath( ci, &aPoint, pathData,
                                            0, 0, false );
            if ( err ) goto bail;

            aPoint.x = 100;
            aPoint.y = 0;
            err = CurveInsertPointIntoPath( ci, &aPoint, pathData,
                                            0, 1, false );
            if ( err ) goto bail;
            aPoint.x = 200;
            aPoint.y = 100;
            err = CurveInsertPointIntoPath( ci, &aPoint, pathData,
                                            0, 2, false );
            if ( err ) goto bail;
            aPoint.x = 100;
            aPoint.y = 200;
            err = CurveInsertPointIntoPath( ci, &aPoint, pathData,
                                            0, 3, false );
            if ( err ) goto bail;
        break;
        case 3:
            aPoint.x = 0;
            aPoint.y = 0;
            err = CurveInsertPointIntoPath( ci, &aPoint, pathData,
                                            0, 0, true );
            if ( err ) goto bail;

            aPoint.x = 200;
            aPoint.y = 50;
            err = CurveInsertPointIntoPath( ci, &aPoint, pathData,
                                            0, 1, true );
            if ( err ) goto bail;
            aPoint.x = 400;
            aPoint.y = 400;
            err = CurveInsertPointIntoPath( ci, &aPoint, pathData,
```

```
                                        0, 2, true );
            if ( err ) goto bail;
        break;
        case 4:
            aPoint.x = Long2Fix( 0 );
            aPoint.y = Long2Fix( 0 );
            err = CurveInsertPointIntoPath( ci, &aPoint, pathData,
                                        0, 0, true );
            if ( err ) goto bail;

            aPoint.x = Long2Fix( 200 );
            aPoint.y = Long2Fix( 50 );
            err = CurveInsertPointIntoPath( ci, &aPoint, pathData,
                                        0, 1, true );
            if ( err ) goto bail;
            aPoint.x = Long2Fix( 400 );
            aPoint.y = Long2Fix( 400 );
            err = CurveInsertPointIntoPath( ci, &aPoint, pathData,
                                        0, 2, true );
            if ( err ) goto bail;
        break;
    }
    err = CurveCreateVectorStream( ci, &vectorData );
    if ( err ) goto bail;
    err = CurveAddPathAtomToVectorStream( ci, pathData, vectorData );
    if ( err ) goto bail;

    err = CurveAddZeroAtomToVectorStream( ci, vectorData );
    if ( err ) goto bail;
bail:
    if ( pathData ) DisposeHandle( pathData );
    if ( ci ) CloseComponent( ci );
    if ( err != noErr ) {
        if ( vectorData ) {
            DisposeHandle( vectorData );
            vectorData = nil;
        }
    }
    return vectorData;
}
```

CreateSamplePathTweenContainer Utility

Listing 7-11 shows how to use the `CreateSamplePathTweenContainer` utility routine.

Listing 7-11 Utility routine CreateSamplePathTweenContainer

```
OSErr CreateSamplePathTweenContainer( QTAtomContainer container,
    OSType tweenerType, long whichSamplePath, Boolean returnDelta,
    TimeValue duration, Fixed initialRotation, QTAtom *newTweenAtom )
{
    OSErr           err = noErr;
    TimeValue       offset;
    Handle          thePathData = nil;
    QTAtom          tweenAtom;
```

```
    err = QTRemoveChildren( container, kParentAtomIsContainer );
    if ( err ) goto bail;
    offset = 0;
    err = AddTweenAtom( container, kParentAtomIsContainer, 1,
                        tweenerType, offset, duration, 0, 0, nil,
                        &tweenAtom );
    if ( err ) goto bail;

    thePathData = CreateSampleVectorData( whichSamplePath );
    if ( thePathData == nil ) { err = memFullErr; goto bail; }

    HLock( thePathData );

    err = AddDataAtom( container, tweenAtom, 1,
                        GetHandleSize( thePathData ), *thePathData,
                        nil, 0, nil );
    if ( err ) goto bail;
    if ( returnDelta ) {
        err = AddPathTweenFlags( container, tweenAtom,
                                    kTweenReturnDelta );
    }
    if ( initialRotation )
        QTInsertChild( container, tweenAtom, kInitialRotationAtom,
                1, 1, sizeof(initialRotation), &initialRotation, nil );
bail:
    if ( thePathData ) DisposeHandle( thePathData );
    if ( newTweenAtom ) *newTweenAtom = tweenAtom;
    return err;
```

Using a kTweenTypePathToMatrixTranslation Tween Component

To use a `kTweenTypePathToMatrixTranslation` tween component, do the following:

1. Create a QT atom container.
2. Insert a `kTweenEntry` atom into the QT atom container for the tween.
3. Insert a `kTweenType` atom that specifies the tween type into the `kTweenEntry` atom.
4. Insert a `kTweenData` atom into the `kTweenEntry` atom.
5. Perform the tweening operation, using `QTDoTween`.

Listing 7-12 shows how to create a `kTweenTypePathToMatrixTranslation` tween.

Listing 7-12 Creating a kTweenTypePathToMatrixTranslation tween container

```
OSErr               err = noErr;
TimeValue           tweenTime, duration;
Handle              result = nil;
QTAtomContainer     container = nil;
QTTweener           tween = nil;
QTAtom              tweenAtom;
```

```
duration = 8;
result = NewHandle( 0 );
if ( err = MemError() ) goto bail;
err = QTNewAtomContainer( &container );
if ( err ) goto bail;
err = CreateSamplePathTweenContainer( container,
                                    kTweenTypePathToMatrixTranslation, 1,
                                    false, duration, 0, &tweenAtom );
if ( err ) goto bail;
err = QTNewTween( &tween, container, tweenAtom, duration );
if ( err ) goto bail;
for ( tweenTime = 0; tweenTime <= duration; tweenTime++ ) {
    MatrixRecord absoluteMatrix;

    err = QTDoTween( tween, tweenTime, result, nil, nil, nil );
    if ( err ) goto bail;

    absoluteMatrix = *(MatrixRecord *)*result;
}

err = QTDisposeTween( tween );
bail:
    if ( container ) QTDisposeAtomContainer( container );
    if ( result ) DisposeHandle( result );
```

Listing 7-13 shows how to create a `kTweenTypePathToMatrixTranslation` tween in which the the `kTweenReturnDelta` flag is set.

Listing 7-13 Creating a kTweenTypePathToMatrixTranslation tween

```
err = CreateSamplePathTweenContainer( container,
                                    kTweenTypePathToMatrixTranslation, 1,
                                    true, duration, 0, &tweenAtom );
if ( err ) goto bail;
err = QTNewTween( &tween, container, tweenAtom, duration );
if ( err ) goto bail;
for ( tweenTime = 0; tweenTime <= duration; tweenTime++ ) {
    MatrixRecord deltaMatrix;

    err = QTDoTween( tween, tweenTime, result, nil, nil, nil );
    if ( err ) goto bail;

    deltaMatrix = *(MatrixRecord *)*result;
}

err = QTDisposeTween( tween );
bail:
    if ( container ) QTDisposeAtomContainer( container );
    if ( result ) DisposeHandle( result );
```

Using a kTweenTypePathToFixedPoint Tween Component

To use a `kTweenTypePathToFixedPoint` tween component, do the following:

1. Create a QT atom container.

2. Insert a `kTweenEntry` atom into the QT atom container for the tween.
3. Insert a `kTweenType` atom that specifies the tween type into the `kTweenEntry` atom.
4. Insert a `kTweenData` atom into the `kTweenEntry` atom.
5. Perform the tweening operation, using `QTDoTween`.

Listing 7-14 shows how to create a `kTweenTypePathToFixedPoint` tween.

Listing 7-14 Creating a kTweenTypePathToFixedPoint tween container

```
err = CreateSamplePathTweenContainer( container,
                                    kTweenTypePathToFixedPoint, 2, false,
                                    duration, 0, &tweenAtom );
if ( err ) goto bail;
err = QTNewTween( &tween, container, tweenAtom, duration );
if ( err ) goto bail;
for ( tweenTime = 0; tweenTime <= duration; tweenTime++ ) {
    gxPoint absolutePoint;

    err = QTDoTween( tween, tweenTime, result, nil, nil, nil );
    if ( err ) goto bail;

    absolutePoint = *(gxPoint *)*result;
}

err = QTDisposeTween( tween );
bail:
    if ( container ) QTDisposeAtomContainer( container );
    if ( result ) DisposeHandle( result );
```

Listing 7-15 shows how to create a `kTweenTypePathToFixedPoint` tween in which the `kTweenReturnDelta` flag is set.

Listing 7-15 Creating a kTweenTypePathToFixedPoint tween container

```
err = CreateSamplePathTweenContainer( container,
                                    kTweenTypePathToFixedPoint, 2, true,
                                    duration, 0, &tweenAtom );
if ( err ) goto bail;
err = QTNewTween( &tween, container, tweenAtom, duration );
if ( err ) goto bail;
for ( tweenTime = 0; tweenTime <= duration; tweenTime++ ) {
    gxPoint deltaPoint;

    err = QTDoTween( tween, tweenTime, result, nil, nil, nil );
    if ( err ) goto bail;

    deltaPoint = *(gxPoint *)*result;
}

err = QTDisposeTween( tween );
bail:
    if ( container ) QTDisposeAtomContainer( container );
    if ( result ) DisposeHandle( result );
```

Using a kTweenTypePathToMatrixRotation Tween Component

To use a `kTweenTypePathToMatrixRotation` tween component, do the following:

1. Create a QT atom container.
2. Insert a `kTweenEntry` atom into the QT atom container for the tween.
3. Insert a `kTweenType` atom that specifies the tween type into the `kTweenEntry` atom.
4. Insert a `kTweenData` atom into the `kTweenEntry` atom.
5. Perform the tweening operation, using `QTDoTween`.

Listing 7-16 shows how to create a `kTweenTypePathToMatrixRotation` tween.

Listing 7-16 Creating a kTweenTypePathToMatrixRotation tween container

```
// kTweenTypePathToMatrixRotation
err = CreateSamplePathTweenContainer( container,
                                kTweenTypePathToMatrixRotation, 1, false,
                                duration, X2Fix(0.5), &tweenAtom );
if ( err ) goto bail;
err = QTNewTween( &tween, container, tweenAtom, duration );
if ( err ) goto bail;
for ( tweenTime = 0; tweenTime <= duration; tweenTime++ ) {
    MatrixRecord absoluteMatrix;

    err = QTDoTween( tween, tweenTime, result, nil, nil, nil );
    if ( err ) goto bail;

    absoluteMatrix = *(MatrixRecord *)*result;
}

err = QTDisposeTween( tween );
bail:
    if ( container ) QTDisposeAtomContainer( container );
    if ( result ) DisposeHandle( result );
```

Using a kTweenTypePathToMatrixTranslationAndRotation Tween Component

To use a `kTweenTypePathToMatrixTranslationAndRotation` tween component, do the following:

1. Create a QT atom container.
2. Insert a `kTweenEntry` atom into the QT atom container for the tween.
3. Insert a `kTweenType` atom that specifies the tween type into the `kTweenEntry` atom.
4. Insert a `kTweenData` atom into the `kTweenEntry` atom.

5. Perform the tweening operation, using `QTDoTween`.

Listing 7-17 shows how to create a `kTweenTypePathToMatrixTranslationAndRotation` tween.

Listing 7-17 Creating a kTweenTypePathToMatrixTranslationAndRotation tween container

```
err = CreateSamplePathTweenContainer( container,
                            kTweenTypePathToMatrixTranslationAndRotation,
                            1, false, duration, X2Fix(0.5), &tweenAtom );
if ( err ) goto bail;
err = QTNewTween( &tween, container, tweenAtom, duration );
if ( err ) goto bail;
for ( tweenTime = 0; tweenTime <= duration; tweenTime++ ) {
    MatrixRecord absoluteMatrix;

    err = QTDoTween( tween, tweenTime, result, nil, nil, nil );
    if ( err ) goto bail;

    absoluteMatrix = *(MatrixRecord *)*result;
}

err = QTDisposeTween( tween );
bail:
    if ( container ) QTDisposeAtomContainer( container );
    if ( result ) DisposeHandle( result );
```

Using a kTweenTypePathXtoY Tween Component

To use `kTweenTypePathXtoY` tween components, either absolute or delta, do the following:

1. Create a QT atom container.
2. Insert a `kTweenEntry` atom into the QT atom container for the tween.
3. Insert a `kTweenType` atom that specifies the tween type into the `kTweenEntry` atom.
4. Insert a `kTweenData` atom into the `kTweenEntry` atom.
5. Perform the tweening operation, using `QTDoTween`.

Listing 7-18 shows how to create both kinds of `kTweenTypePathXtoY` tweens.

Listing 7-18 Creating kTweenTypePathXtoY tweens container

```
// kTweenTypePathXtoY - normal
err = CreateSamplePathTweenContainer( container, kTweenTypePathXtoY, 3,
                                        false, duration, 0, &tweenAtom );
if ( err ) goto bail;
err = QTNewTween( &tween, container, tweenAtom, duration );
if ( err ) goto bail;
for ( tweenTime = 0; tweenTime <= duration; tweenTime++ ) {
    Fixed absoluteYvalue;
```

```
    err = QTDoTween( tween, tweenTime, result, nil, nil, nil );
    if ( err ) goto bail;

    absoluteYvalue = *(Fixed *)*result;
}

err = QTDisposeTween( tween );

// kTweenTypePathXtoY - delta
err = CreateSamplePathTweenContainer( container, kTweenTypePathXtoY, 3,
                                        true, duration, 0, &tweenAtom );
if ( err ) goto bail;
err = QTNewTween( &tween, container, tweenAtom, duration );
if ( err ) goto bail;
for ( tweenTime = 0; tweenTime <= duration; tweenTime++ ) {
    Fixed deltaYalue;

    err = QTDoTween( tween, tweenTime, result, nil, nil, nil );
    if ( err ) goto bail;

    deltaYalue = *(Fixed *)*result;
}

err = QTDisposeTween( tween );
bail:
    if ( container ) QTDisposeAtomContainer( container );
    if ( result ) DisposeHandle( result );
```

Using a kTweenTypePathYtoX Tween Component

To use `kTweenTypePathYtoX` tween components, either absolute or delta, do the following:

1. Create a QT atom container.
2. Insert a `kTweenEntry` atom into the QT atom container for the tween.
3. Insert a `kTweenType` atom that specifies the tween type into the `kTweenEntry` atom.
4. Insert a `kTweenData` atom into the `kTweenEntry` atom.
5. Perform the tweening operation, using `QTDoTween`.

Listing 7-19 shows how to create both kinds of `kTweenTypePathYtoX` tweens.

Listing 7-19 Creating kTweenTypePathYtoX tweens container

```
// kTweenTypePathYtoX - normal
err = CreateSamplePathTweenContainer( container, kTweenTypePathYtoX, 4,
                                        false, duration, 0, &tweenAtom );
if ( err ) goto bail;
err = QTNewTween( &tween, container, tweenAtom, duration );
if ( err ) goto bail;
for ( tweenTime = 0; tweenTime <= duration; tweenTime++ ) {
```

```
        Fixed absoluteXvalue;

        err = QTDoTween( tween, tweenTime, result, nil, nil, nil );
        if ( err ) goto bail;

        absoluteXvalue = *(Fixed *)*result;
    }

    err = QTDisposeTween( tween );
    // kTweenTypePathYtoX - delta
    err = CreateSamplePathTweenContainer( container, kTweenTypePathYtoX, 4,
                                        true, duration, 0, &tweenAtom );
    if ( err ) goto bail;
    err = QTNewTween( &tween, container, tweenAtom, duration );
    if ( err ) goto bail;
    for ( tweenTime = 0; tweenTime <= duration; tweenTime++ ) {
        Fixed deltaXvalue;

        err = QTDoTween( tween, tweenTime, result, nil, nil, nil );
        if ( err ) goto bail;

        deltaXvalue = *(Fixed *)*result;
    }

    err = QTDisposeTween( tween );
    bail:
        if ( container ) QTDisposeAtomContainer( container );
        if ( result ) DisposeHandle( result );
```

Tween Components and Native Tween Types

This chapter describes the native tween types handled by QuickTime; the tween components included in QuickTime; and the constants, data types, and routines associated with tween components.

Each component processes one or more input values contained in the tween media and returns output values.

Tween QT Atom Container

The characteristics of a tween are specified by the atoms in a tween QT atom container. A tween QT atom container can contain the atom types described in this section.

General Tween Atoms

The `kTweenEntry` atom specifies a tween that can be either a single tween, a tween in a tween sequence, or an interpolation tween. Its parent is the tween QT atom container (which you specify with the constant `kParentAtomIsContainer`).

The index of a `kTweenEntry` atom specifies when it was added to the QT atom container; the first added has the index 1, the second 2, and so on. The ID of a `kTweenEntry` atom can be any ID that is unique among the `kTweenEntry` atoms contained in the same QuickTime atom container.

This atom is a parent atom. It must contain the following child atoms:

- A `kTweenType` atom that specifies the tween type.
- One or more `kTweenData` atoms that contain the data for the tween. Each `kTweenData` atom can contain different data to be processed by the tween component, and a tween component can process data from only one `kTweenData` atom a time. For example, an application can use a list tween to animate sprites. The `kTweenEntry` atom for the tween could contain three sets of animation data, one for moving the sprite from left to right, one for moving the sprite from right to left, and one for moving the sprite from top to bottom. In this case, the `kTweenEntry` atom for the tween would contain three `kTweenData` atoms, one for each data set. The application specifies the desired data set by specifying the ID of the `kTweenData` atom to use.

A `kTweenEntry` atom can contain any of the following optional child atoms:

- A `kTweenStartOffset` atom that specifies a time interval, beginning at the start of the tween media sample, after which the tween operation begins. If this atom is not included, the tween operation begins at the start of the tween media sample.
- A `kTweenDuration` atom that specifies the duration of the tween operation. If this atom is not included, the duration of the tween operation is the duration of the media sample that contains it.

If this atom specifies a path tween, it can contain the following optional child atom:

- A kTweenFlags atom containing flags that control the tween operation. If this atom is not included, no flags are set.

If a kTweenEntry atom specifies an interpolation tween, it must contain the following child atom(s):

- A kTweenInterpolationID atom for each kTweenData atom to be interpolated. The ID of each kTweenInterpolationID atom must match the ID of the kTweenData atom to be interpolated. The data for a kTweenInterpolationID atom specifies a kTweenEntry atom that contains the interpolation tween to use for the kTweenData atom.

If this atom specifies an interpolation tween, it can contain either of the following optional child atoms:

- A kTweenOutputMin atom that specifies the minimum output value of the interpolation tween. The value of this atom is used only if there is also a kTweenOutputMax atom with the same parent. If this atom is not included and there is a kTweenOutputMax atom with the same parent, the tween component uses 0 as the minimum value when scaling output values of the interpolation tween.
- A kTweenOutputMax atom that specifies the maxiumum output value of the interpolation tween. If this atom is not included, the tween component does not scale the output values of the interpolation tween.
- A kTweenStartOffset atom. For a tween in a tween track of a QuickTime movie, this atom specifies a time offset from the start of the tween media sample to the start of the tween. The time units are the units used for the tween track. Its parent atom is a kTweenEntry atom.

A kTweenEntry atom can contain only one kTweenStartOffset atom. The ID of this atom is always 1. The index of this atom is always 1.

This atom is a leaf atom. The data type of its data is TimeValue.

This atom is optional. If it is not included, the tween operation begins at the start of the tween media sample.

The kTweenDuration atom specifies the duration of a tween operation. When a QuickTime movie includes a tween track, the time units for the duration are those of the tween track. If a tween component is used outside of a movie, the application using the tween data determines how the duration value and values returned by the component are interpreted. Its parent atom is a kTweenEntry atom.

A kTweenEntry atom can contain only one kTweenDuration atom. The ID of this atom is always 1. The index of this atom is always 1.

This atom is a leaf atom. The data type of its data is TimeValue.

This atom is optional. If it is not included, the duration of the tween is the duration of the media sample that contains it.

The kTweenData atom contains data for a tween. Its parent atom is a kTweenEntry atom.

A kTweenEntry atom can contain any number of kTweenData atoms. Each kTweenData atom can contain different data to be processed by the tween component, and a tween component can process data from only one kTweenData atom a time. For example, an application can use a list tween to animate sprites. The kTweenEntry atom for the tween could contain three sets of animation data, one for moving the sprite from left to right, one for moving the sprite from right to left, and one for moving the sprite from top to bottom. In this case, the kTweenEntry atom for the tween would contain three kTweenData atoms, one for each data set. The application would specify the desired data set by specifying the ID of the kTweenData atom to use.

The index of a kTweenData atom specifies when it was added to the kTweenEntry atom; the first added has the index 1, the second 2, and so on. The ID of a kTweenData atom can be any ID that is unique among the kTweenData atoms contained in the same kTweenEntry atom.

At least one kTweenData atom is required in a kTweenEntry atom.

For single tweens, a kTweenData atom is a leaf atom. It can contain data of any type.

For polygon tweens, a kTweenData atom is a leaf atom. The data type of its data is Fixed[27], which specifies three polygons as described in Using Path Tween Components (page 36).

For path tweens, a kTweenData atom is a leaf atom. The data type of its data is Handle, which contains a QuickTime vector as described in Using Path Tween Components (page 36).

In interpolation tweens, a kTweenData atom is a leaf atom. It can contain data of any type. An interpolation tween can be any tween other than a list tween that returns a time value, as described in Interpolation Tweens (page 26).

In list tweens, a kTweenData atom is a parent atom that must contain the following child atoms:

- A kListElementType atom that specifies the atom type of the elements iof the tween.
- One or more leaf atoms of the type specified by the kListElementType atom. The data for each atom is the result of a list tween operation, as described in Using a List Tween Component (page 36).

The kNameAtom atom specifies the name of a tween. Its parent atom is a kTweenEntry atom. The name, which is optional, is not used by tween components, but it can be used by applications or other software.

A kTweenEntry atom can contain only one kNameAtom atom. The ID of this atom is always 1. The index of this atom is always 1.

This atom is a leaf atom. Its data type is StringPtr.

This atom is optional. If it is not included, the tween does not have name.

The kTweenType atom specifies the tween type (the data type of the data for the tween operation). Its parent atom is a kTweenEntry atom.

A kTweenEntry atom can contain only one kTweenType atom. The ID of this atom is always 1. The index of this atom is always 1.

This atom is a leaf atom. The data type of its data is OSType.

This atom is required.

Path Tween Atoms

The kTweenFlags atom contains flags that control the tween operation. Its parent atom is a kTweenEntry atom. One flag that controls path tweens is defined: the kTweenReturnDelta flag. It applies only to path tweens (tweens of type kTweenTypePathToFixedPoint, kTweenTypePathToMatrixTranslation, kTweenTypePathToMatrixTranslationAndRotation, kTweenTypePathXtoY, or kTweenTypePathYtoX). If the flag is set, the tween component returns the change in value from the last time it was invoked. If the flag is not set, or if the tween component has not previously been invoked, the tween component returns the normal result for the tween.

A kTweenEntry atom can contain only one kTweenFlags atom. The ID of this atom is always 1. The index of this atom is always 1.

This atom is a leaf atom. The data type of its data is Long.

This atom is optional. If it is not included, no flags are set.

The kInitialRotationAtom atom specifies an initial angle of rotation for a path tween of type kTweenTypePathToMatrixRotation, kTweenTypePathToMatrixTranslation, or kTweenTypePathToMatrixTranslationAndRotation. Its parent atom is a kTweenEntry atom.

A kTweenEntry atom can contain only one kInitialRotationAtom atom. The ID of this atom is always 1. The index of this atom is always 1.

This atom is a leaf atom. Its data type is Fixed.

This atom is optional. If it is not included, no initial rotation of the tween is performed.

List Tween Atoms

The kListElementType atom specifies the atom type of the elements in a list tween. Its parent atom is a kTweenData atom.

A kTweenEntry atom can contain only one kListElementType atom. The ID of this atom is always 1. The index of this atom is always 1.

This atom is a leaf atom. Its data type is QTAtomType.

This atom is required in the kTweenData atom for a list tween.

Interpolation Tween Atoms

The kTweenOutputMax atom specifies the maximum output value of an interpolation tween. Its parent atom is a kTweenEntry atom.

If a kTweenOutputMax atom is included for an interpolation tween, output values for the tween are scaled to be within the minimum and maximum values. The minimum value is either the value of the kTweenOutputMin atom or, if there is no kTweenOutputMin atom, 0. For example, if an interpolation tween has values between 0 and 4, and it has kTweenOutputMin and kTweenOutputMax atoms with values 1 and 2, respectively, a value of 0 (the minimum value before scaling) is scaled to 1 (the minimum specified by the kTweenOutputMin atom), a value of 4 (the maximum value before scaling) is scaled to 2 (the maximum specified by the kTweenOutputMax atom), and a value of 3 (three-quarters of the way between the maximum and minimum values before scaling) is scaled to 1.75 (three-quarters of the way between the values of the kTweenOutputMin and kTweenOutputMax atoms).

A kTweenEntry atom can contain only on e kTweenOutputMax atom. The ID of this atom is always 1. The index of this atom is always 1.

This atom is a leaf atom. The data type of its data is `Fixed`.

This atom is optional. If it is not included, QuickTime does not scale interpolation tween values.

The `kTweenOutputMin` atom specifies the minimum output value of an interpolation tween. Its parent atom is a `kTweenEntry` atom.

If both `kTweenOutputMin` and `kTweenOutputMax` atoms are included for an interpolation tween, output values for the tween are scaled to be within the minimum and maximum values. For example, if an interpolation tween has values between 0 and 4, and it has `kTweenOutputMin` and `kTweenOutputMax` atoms with values 1 and 2, respectively, a value of 0 (the minimum value before scaling) is scaled to 1 (the minimum specified by the `kTweenOutputMin` atom), a value of 4 (the maximum value before scaling) is scaled to 2 (the maximum specified by the `kTweenOutputMax` atom), and a value of 3 (three-quarters of the way between the maximum and minimum values before scaling) is scaled to 1.75 (three-quarters of the way between the values of the `kTweenOutputMin` and `kTweenOutputMax` atoms).

If a `kTweenOutputMin` atom is included but a `kTweenOutputMax` atom is not, QuickTime does not scale interpolation tween values.

A `kTweenEntry` atom can contain only on e `kTweenOutputMin` atom. The ID of this atom is always 1. The index of this atom is always 1.

This atom is a leaf atom. The data type of its data is `Fixed`.

This atom is optional. If it is not included but a `kTweenOutputMax` atom is, the tween component uses `0` as the minimum value for scaling values of an interpolation tween.

The `kTweenInterpolationID` atom specifies an interpolation tween to use for a specified `kTweenData` atom. Its parent atom is a `kTweenEntry` atom. There can be any number of `kTweenInterpolationID` atoms for a tween, one for each `kTweenData` atom to be interpolated.

The index of a `kTweenInterpolationID` atom specifies when it was added to the `kTweenEntry` atom; the first added has the index 1, the second 2, and so on. The ID of a `kTweenInterpolationID` atom must (1) match the atom ID of the `kTweenData` atom to be interpolated, and (2) be unique among the `kTweenInterpolationID` atoms contained in the same `kTweenEntry` atom.

This atom is a leaf atom. The data type of its data is `QTAtomID`.

This atom is required for an interpolation tween.

Sequence Tween Atoms

The `kTweenSequenceElement` atom specifies an entry in a tween sequence. Its parent is the tween QT atom container (which you specify with the constant `kParentAtomIsContainerTween` Components and Tween Media).

The ID of a `kTweenSequenceElement` atom must be unique among the `kTweenSequenceElement` atoms in the same QT atom container. The index of a `kTweenSequenceElement` atom specifies its order in the sequence; the first entry in the sequence has the index 1, the second 2, and so on.

This atom is a leaf atom. The data type of its data is `TweenSequenceEntryRecord`, a data structure that contains the following fields:

- endPercent, a value of type Fixed that specifies the point in the duration of the tween media sample at which the sequence entry ends. This is expressed as a percentage; for example, if the value is 75.0, the sequence entry ends after three-quarters of the total duration of the tween media sample have elapsed. The sequence entry begins after the end of the previous sequence entry or, for the first entry in the sequence, at the beginning of the tween media sample.
- tweenAtomID, a value of type QTAtomID that specifies the kTweenEntry atom containing the tween for the sequence element. The kTweenEntry atom and the kTweenSequenceElement atom must both be a child atoms of the same tween QT atom container.
- dataAtomID, a value of type QTAtomID that specifies the kTweenData atom containing the data for the tween. This atom must be a child atom of the atom specified by the tweenAtomID field.

Tween Container Syntax

Tween containers conform to the following syntax:

```
[(TweenContainerFormat)] = [(SingleTweenFormat)] | [(SequencedTweenFormat)]

[(SingleTweenFormat)] = [(TweenEntryAtoms)] <kTweenEntry>, (anyUniqueIDs),
    (1..numInterpolators)

[(TweenEntryAtoms)] [(SequencedTweenFormat)] = kTweenSequenceElement,
    (anyUniqueIDs), (1..numSequenceElements)

[TweenSequenceEntryRecord] = {endPercent, tweenAtomID, dataAtomID} kTweenEntry,
    (anyUniqueIDs), (1..numSequenceElements + numInterpolators)

[(TweenEntryAtoms)] [(TweenEntryAtoms)] = kTweenType, 1, 1

[OSType] = the type of tween <kTweenStartOffset>, 1, 1

[TimeValue] = starting offset <kTweenDuration>, 1, 1

[TimeValue] = duration <kTweenOutputMinValue>, 1, 1

[Fixed] = minimum output value <kTweenOutputMaxValue>, 1, 1

[Fixed] = maximum output value <kTweenFlags>, 1, 1

[long] = flags kTweenData, (anyUniqueIDs), (1..numDataAtoms)
 // contents dependent on kTweenType, could be leaf data or nested atoms
    <kTweenInterpolationID>, (a kTweenData ID), (1.. numInterpolationIDAtoms)

[QTAtomID] = the id of a kTweenEntry (child of [(TweenContainerFormat)]
    describing the tween to be used to interpolate time values).
```

kTweenTypeFixed

Input data: Two 32-bit fixed-point values.

Output data: A 32-bit fixed-point value.

kTweenTypeFixedPoint

Input data: Two structures of type `FixedPoint` that describe QuickDraw points.

Output data: A structure of type `FixedPoint` that describes a QuickDraw point.

How interpolation is performed: Each of the two coordinate values used to specify a fixed point is interpolated separately from the other.

kTweenTypeGraphicsModeWithRGBColor

Input data: Two `ModifierTrackGraphicsModeRecord` data structures.

Output data: A `ModifierTrackGraphicsModeRecord` data structure.

How interpolation is performed: Only the `RGBColor` fields of the `ModifierTrackGraphicsModeRecord` data structures are interpolated. The graphic mode used is the first graphic mode that is specified in the modifier track.

kTweenTypeLong

Input data: Two signed 32-bit integers.

Output data: A signed 32-bit integer.

kTweenTypeMatrix

Input data: Two QuickTime 3X3 matrices (data structures of type `MatrixRecord`).

Output data: A QuickTime 3X3 matrix (a data structure of type `MatrixRecord`).

How interpolation is performed: Each of the individual matrix elements is interpolated separately from the other.

kTweenTypePoint

Input data: Two QuickDraw points (data structures of type `Point`.

Output data: A QuickDraw point (a data structure of type `Point`).

How interpolation is performed: Each of the two coordinate values used to specify a QuickDraw point (h and `v`) is interpolated separately from the other.

kTweenTypeQTFloatDouble

Input data: Two double-precision (64-bit) IEEE floating-point numbers of type `double`.

Output data: A double-precision (64-bit) IEEE floating-point number of type `double`.

kTweenTypeQTFloatSingle

Input data: Two single-precision floating-point numbers of type `float`.

Output data: A single-precision floating-point number of type `float`.

kTweenTypeRGBColor

Input data: Two RGB colors (data structures of type `RGBColor`).

Output data: An RGB color (a data structure of type `RGBColor`).

How interpolation is performed: Each of the three values used to specify an RGB color (red, green, blue) is interpolated separately from the others.

kTweenTypeShort

Input data: Two signed 16-bit integers.

Output data: A signed 16-bit integer.

Other Tween Components

QuickTime includes a number of other components for processing tweens. These components are described in the following sections. Each component processes one or more input values contained in the tween media and returns output values. The description of each tween component lists the data types of the component's input and output data and the number of input values that the component processes.

List Tweener Components

A component of type `kTweenTypeAtomList` is the component that processes list tweens. For an introduction to list tweens, see Using a List Tween Component (page 36).

Input data: A QT atom list. This is a `kTweenData` atom that contains

- a `kListElementType` atom that specifies the atom type of the elements and the output
- one or more leaf atoms, ordered by their index values, of the type specified by the `kListElementType` atom
- atoms of other types, which are optional and ignored by the component; these optional atoms can be used by an application, such as atoms that specify a name for each element.

Output data: The data for one of the list element atoms. How this atom is determined is described in Using a List Tween Component (page 36).

How interpolation is performed: The duration of the tween is divided by the number of elements in the list. At the time point for which a result is to be returned, the component determines the list element for which to return data. If (0 <= *time value* <= (1 * (*tween duration / number of list elements)),* the component returns the data for the first list element; if ((1 * (*tween duration / number of list elements < time value* <= (2 * (*tween duration / number of list elements)),* the component returns the data for the second list element, and so on. For example, if the tween duration is 100 and there are 10 elements in the list, the component returns the value of the first element for a time from 0 to 10, the value of the second element is returned for a time greater than 10 and less than or equal to 20, and so on. The total time for the tween is divided equally among the list elements.

The `kTweenTypeAtomList` container description is the following:

```
[(QTAtomListTweenEntryAtoms)] =
    kListElementType, 1, 1
    // a QTAtomType specifying the type of atoms that make up the list
    kListElementDataType, 1, 1
    // a UInt32 that contains one of the allowed kTweenType flags.
    // kTweenTypeShort through kTweenTypeFixedPoint are allowed]
    kTweenData, 1, 1
        kTweenType, 1, 1
        // QTAtomType for elements in list, for example 'pcid'
        'pcid', anyUniqueID, 1
            [data for first element]
        'pcid', anyUniqueID, 2
        // data for second element
        ...
        'pcid', anyUniqueID, n
        // data for nth element

<kTweenStartOffset>, 1, 1
[TimeValue] = starting offset

<kTweenDuration>, 1, 1
[TimeValue] = duration

<kTweenOutputMinValue>, 1, 1
[Fixed] = minimum output value

<kTweenOutputMaxValue>, 1, 1
[Fixed] = maximum output value

<kTweenSequenceElement>, (anyUniqueIDs), (1..numElementsInSequence)
[TweenSequenceEntryRecord]

<kTweenInterpolationID>, (a kTweenData ID), (1.. numInterpolationIDAtoms)
[QTAtomID] = the id of a kTweenEntry (child of [(TweenContainerFormat)]
       describing the tween to be used to interpolate time values).
```

Multimatrix Tweener Component

The multimatrix tweener, of type `kTweenTypeMultiMatrix`, returns a `MatrixRecord` that is a concatenation of several matrix tweeners. This record can be applied to a sprite or track.

An example of using the multimatrix tweener would be to make a sprite follow a path using the path tweener and at the same time apply a distortion effect using the polygon tweener.

Input data: A list of `kTweenEntry` atoms.

Output data: a matrix record

How interpolation is performed: The data for the tweener consists of a list of `kTweenEntry` atoms, each containing [(`QTAtomListEntryAtomsTween` Components and Tween Media)] for any type of tweener that returns a matrix. The order of matrix concatenation is important; the matrices are applied in the order determined by index of the `kTweenEntry` child atoms of the multimatrix tweener's data atom.

Path Tweener Components

A path tweener component returns a point along a path depending on the current time value. The point is either returned as a `FixedPoint` value or a `MatrixRecord` with *x* and *y* offsets corresponding to the point. There are six component subtypes:

- Subtype `kTweenTypePathToFixedPoint` returns a `tweenResult` of type `FixedPoint`.
- Subtype `kTweenTypePathToMatrixTranslation` returns a `tweenResult` of type `MatrixRecord` and performs translation tweening.
- Subtype `kTweenTypePathToMatrixRotation` returns a `tweenResult` of type `MatrixRecord` and performs rotation tweening.
- Subtype `kTweenTypePathToMatrixTranslationAndRotation` returns a `tweenResult` of type `MatrixRecord` and performs both translation and rotation tweening.
- Subtype `kTweenTypePathXtoY` returns a `tweenResult` of type `FixedPoint` that specifies the y-coordinate value for a given x-coordinate value.
- Subtype `kTweenTypePathYtoX` returns a `tweenResult` of type `FixedPoint` that specifies the x-coordinate value for a given y-coordinate value.

An example of using a path tweener would be to store a path that you want a sprite to follow. The `MatrixRecord` returned could be used to determine the offset of the sprite.

The path tweener's path format is the one used by the QuickTime vector codec. A transcoder exists that converts a QuickDraw GX shape into this format.

This component uses only the first contour of the first path in the vector data when determining the output. It ignores any additional atoms and paths in the vector data.

Note: If the `kTweenReturnDelta` flag (in an optional `kTweenFlags` atom) is set, the component returns the change in value from the last time it was invoked. If the flag is not set, or if the component has not previously been invoked, the component returns the normal result for the tween.

Polygon Tweener Component

A component of type `kTweenTypePolygon` tweens one polygon into another. All input polygons must be convex and not self-intersecting.

Input data: An array of 27 fixed-point values (`Fixed[27]Tween` Components and Tween Media) that specifies three four-sided polygons. Each polygon is specified by 9 consecutive array elements. The first element in each set of 9 contains the number of points used to specify the polygon; this value is coerced to a long integer, and it must always be 4 after coercion. The following 8 values in each set of nine are four x, y pairs that specify the corners of the polygon.

The first set of 9 elements specifies the dimensions of a sprite or track to be mapped. For example, if the object is a sprite, the four points are (0,0), (*spriteWidth,* 0), (*spriteWidth, spriteHeight),* (0, *spriteHeight).* The next set of 9 elements specifies the initial polygon into which the sprite or track is mapped. The next set of 9 elements specifies the ending polygon into which the sprite or track is mapped.

Output data: A `MatrixRecord` structure that can be used to map the sprite or track into a four-sided polygon. During the duration of the tween, the shape of this polygon is transformed linearly from that of the initial polygon specified in the input data to that of the ending polygon specified in the input data.

Spin Tweener Component

A component of type `kTweenTypeSpin` returns a `MatrixRecord` that can be applied to a sprite or a track. The matrix returned causes a rotation based on a given number of rotations over the duration of the tween. The data for the tweener consists of an array of two `Fixed` numbers. The first `Fixed` number is the `intialRotation` value, specified as a fraction of one rotation. A number between 0 and 1 is expected; for instance, a value of 0.25 represents a rotation of 90 degrees. The second `Fixed` number is the number of rotations that should occur over the durationof the tween. For instance, to spin a sprite four and a half times this number should be 4.5.

Note: The rotation is performed about an object's origin, which is usually 0.0. For a sprite, its origin is defined by its registration point; hence a spinning sprite will spin about its registration point.

The spin tweener container description is the following:

```
[SpinTweenEntryAtoms)] =
    kTweenType, 1, 1
        [kTweenTypeSpin]
    kTweenData, 1, 1
        Fixed[2]
<kTweenStartOffset>, 1, 1
[TimeValue] = starting offset

<kTweenDuration>, 1, 1
[TimeValue] = duration

<kTweenSequenceElement>, (anyUniqueIDs), (1..numElementsInSequence)
[TweenSequenceEntryRecord]

<kTweenInterpolationID>, (a kTweenData ID), (1.. numInterpolationIDAtoms)
[QTAtomID] = the id of a kTweenEntry (child of [(TweenContainerFormat)]
              describing the tween to be used to interpolate time values).
```

Constants

This section defines the constants used with tween components. Included are a variety of tween atom types, as well as the flags used to control tween components.

Tween Component Constant

The TweenComponentType constant specifies that the component is a tween component.

```
enum {
    TweenComponentType  = 'twen'
};
```

Tween Type and Tween Component Subtype Constants

The following constants specify tween types. If a tween type is identified by a four-character code, the four-character code is also the component subtype of the tween component that is invoked for the tween. If a tween type is identified by a numeric constant, such as `kTweenTypeShort`, tweens of that type are processed by QuickTime rather than by a tween component.

```
enum {
    kTweenTypeShort = 1,
    kTweenTypeLong = 2,
    kTweenTypeFixed = 3,
    kTweenTypePoint = 4,
    kTweenTypeQDRect = 5,
    kTweenTypeQDRegion = 6,
    kTweenTypeMatrix = 7,
    kTweenTypeRGBColor = 8,
    kTweenTypeGraphicsModeWithRGBColor = 9,
    kTweenTypeQTFloatSingle = 10,
    kTweenTypeQTFloatDouble = 11,
    kTweenTypeFixedPoint = 12,
    kTweenType3dScale = FOUR_CHAR_CODE('3sca'),
    kTweenType3dTranslate = FOUR_CHAR_CODE('3tra'),
    kTweenType3dRotate = FOUR_CHAR_CODE('3rot'),
    kTweenType3dRotateAboutPoint = FOUR_CHAR_CODE('3rap'),
    kTweenType3dRotateAboutAxis = FOUR_CHAR_CODE('3rax'),
    kTweenType3dRotateAboutVector = FOUR_CHAR_CODE('3rvc'),
    kTweenType3dQuaternion = FOUR_CHAR_CODE('3qua'),
    kTweenType3dMatrix = FOUR_CHAR_CODE('3mat'),
    kTweenType3dCameraData = FOUR_CHAR_CODE('3cam'),
    kTweenType3dSoundLocalizationData = FOUR_CHAR_CODE('3slc'),
    kTweenTypePathToMatrixTranslation = FOUR_CHAR_CODE('gxmt'),
    kTweenTypePathToMatrixTranslationAndRotation =FOUR_CHAR_CODE('gxmr'),
    kTweenTypePathToFixedPoint = FOUR_CHAR_CODE('gxfp'),
    kTweenTypePathXtoY = FOUR_CHAR_CODE('gxxy'),
    kTweenTypePathYtoX = FOUR_CHAR_CODE('gxyx'),
    kTweenTypeAtomList = FOUR_CHAR_CODE('atom'),
    kTweenTypePolygon = FOUR_CHAR_CODE('poly')
    kTweenTypePathToMatrixRotation = FOUR_CHAR_CODE('gxpr'),
    kTweenTypeMultiMatrix = FOUR_CHAR_CODE('mulm'),
```

```
    kTweenTypeSpin = FOUR_CHAR_CODE('spin'),
    kTweenType3dMatrixNonLinear = FOUR_CHAR_CODE('3nlr'),
    kTweenType3dVRObject = FOUR_CHAR_CODE('3vro')
    };
```

Tween Atom Constants

The following constants are defined for tween-related atoms. These atom types are described in Tween QT Atom Container (page 57).

```
enum {
    kTweenEntry                 = FOUR_CHAR_CODE('twen'),
    kTweenData                  = FOUR_CHAR_CODE('data'),
    kTweenType                  = FOUR_CHAR_CODE('twnt'),
    kTweenStartOffset           = FOUR_CHAR_CODE('twst'),
    kTweenDuration              = FOUR_CHAR_CODE('twdu'),
    kTweenFlags                 = FOUR_CHAR_CODE('flag'),
    kTweenOutputMin             = FOUR_CHAR_CODE('omin'),
    kTweenOutputMax             = FOUR_CHAR_CODE('omax'),
    kTweenSequenceElement       = FOUR_CHAR_CODE('seqe'),
    kTween3dInitialCondition    = FOUR_CHAR_CODE('icnd'),
    kTweenInterpolationID       = FOUR_CHAR_CODE('intr'),
    kTweenRegionData            = FOUR_CHAR_CODE('qdrg'),
    kTweenPictureData           = FOUR_CHAR_CODE('PICT'),
    kListElementType            = FOUR_CHAR_CODE('type'),
    kNameAtom                   = FOUR_CHAR_CODE('name'),
    kInitialRotationAtom        = FOUR_CHAR_CODE('inro')
};
```

Tween Flag

The following flag modifies the characteristics of a tween.

```
enum {
    kTweenReturnDelta= 1L << 0
};
```

The `kTweenReturnDelta` flag applies only to path tweens (tweens of type `kTweenTypePathToFixedPoint`, `kTweenTypePathToMatrixTranslation`, `kTweenTypePathToMatrixTranslationAndRotation`, `kTweenTypePathXtoY`, or `kTweenTypePathYtoX`). If the flag is set, the tween component returns the change in value from the last time it was invoked. If the flag is not set, or if the tween component has not previously been invoked, the tween component returns the normal result for the tween.

Data Types

The following sections describe the component instance definition, tween record, and the value setting prototype function used by tween components.

Tween Sequence Entry Record

A tween sequence entry record specifies when a tween in a tween sequence ends. Each tween in a tween sequence begins after the previous tween ends or, for the first tween in the sequence, at the beginning of the tween duration.

Because there can be more than one data set for a tween, the data structure includes a field for the data atom ID as well as the tween atom ID.

```
struct TweenSequenceEntryRecord {
    Fixed                   endPercent;
    QTAtomID                tweenAtomID;
    QTAtomID                dataAtomID;
};
typedef struct TweenSequenceEntryRecord TweenSequenceEntryRecord;
```

Term	Definition
endPercent	a value of type Fixed that specifies the point in the duration of the tween media sample at which the sequence entry ends.
tweenAtomID	a value of type QTAtomID that specifies the kTweenEntry atom containing the tween for the sequence element.
dataAtomID	a value of type QTAtomID that specifies the kTweenData atom containing the data for the tween.

Component Instance

The component instance for a tween component, TweenerComponent, identifies an application's use of a component.

```
typedef ComponentInstance TweenerComponent;
```

Tween Record

QuickTime maintains a tween record structure that is provided to your tween component's TweenDoTween method. The TweenRecord structure is defined as follows.

```
typedef struct TweenRecord TweenRecord;

struct TweenRecord {
    long                    version;
    QTAtomContainer         container;
    QTAtom                  tweenAtom;
    QTAtom                  dataAtom;
    Fixed                   percent;
    TweenerDataUPP          dataProc;
    void *                  private1;
    void *                  private2;
};
```

Term	Definition
version	The version number of this structure. This field is initialized to 0.
container	The atom container that contains the tween data.
tweenAtom	The atom for this tween entry's data in the container.
percent	The percentage by which to change the data.
dataProc	The procedure the tween component calls to send the tweened value to the receiving track.
private1	Reserved.
private2	Reserved.

Value Setting Function

The function that you call to send the interpolated value to the receiving track is defined as a universal procedure in systems that support the Macintosh Code Fragment Manager (CFM) or is defined as a data procedure for non-CFM systems:

```
typedef UniversalProcPtr TweenerDataUPP;    /* CFM */

typedef TweenerDataProcPtr TweenerDataUPP;  /* non-CFM */
```

The TweenerDataUPP function pointer specifies the function the tween component calls with the value generated by the tween operation. A tween component calls this function from its implementation of the TweenerDoTween function.

```
typedef pascal ComponentResult (*TweenerDataProcPtr)(
                     TweenRecord *tr,
                     void *tweenData,
                     long tweenDataSize,
                     long dataDescriptionSeed,
                     Handle dataDescription,
                     ICMCompletionProcRecordPtr asyncCompletionProc,
                     ProcPtr transferProc,
                     void *refCon);
```

Term	Definition
tr	Contains a pointer to the tween record for the tween operation.
tweenData	Contains a pointer to the generated tween value.
tweenDataSize	Specifies the size, in bytes, of the tween value.
dataDescriptionSeed	Specifies the starting value for the calculation. Every time the content of the dataDescription handle changes, this value should be incremented.

Term	Definition
dataDescription	Specifies a handle containing a description of the tween value passed. For basic types such as integers, the calling tween component should set this parameter to nil. For more complex types such as compressed image data, the calling tween component should set this handle to contain a description of the tween value, such as an image description.
asyncCompletionProc	Contains a pointer to a completion procedure for asynchronous operations. The calling tween component should set the value of this parameter to `nil`.
transferProc	Contains a pointer to a procedure to transfer the data. The calling tween component should set the value of this parameter to `nil`.
refCon	Contains a pointer to a reference constant. The calling tween component should set the value of this parameter to `nil`.

You call this function by invoking the function specified in the tween record's `dataProc` field. The following errors are returned:

Constant	Value	Description
noErr	0	No error
paramErr	-50	Invalid parameter specified

Creating a Tween Component

This chapter explains how to create a tween component for a new data type, a new interpolation algorithm, or both. Before reading this section, you should be familiar with how to create components.

The following example illustrates a tween component that interpolates values for short integers. Because QuickTime handles this tween type (`kTweenTypeShortTween` Components and Tween Media) for you, you do not need to implement a component to handle interpolation of short integers yourself.

Initializing the Tween Component

Your tween component must process `kTweenerInitializeSelect` requests from the Component Manager. Listing 9-1 shows a function, `TweenerInitialize`, for processing this request. In this example, the function simply returns. In a more complex example, the function might allocate storage to be used when generating a tween media value.

Listing 9-1 Function that initializes a tween component

```
pascal ComponentResult TweenerInitialize (
                                             TweenerComponent tc,
                                             QTAtomContainer container,
                                             QTAtom tweenAtom,
                                             QTAtom dataAtom)
{
    return noErr;
}
```

Generating Tween Media Values

Your tween component must process `kTweenerDoTweenSelect` requests from the Component Manager. Listing 9-2 shows a function, `TweenDoTween`, for processing this request. It takes short-integer values and performs the necessary interpolation.

Listing 9-2 Function that generates tween media values

```
pascal ComponentResult TweenDoTween (
                                         TweenerComponent tc,
                                         TweenRecord *tr)
{
    short       *data;
    short       tFrom, tTo, tValue;
    QTGetAtomDataPtr(tr->container, tr->dataAtom, nil, (Ptr *)&data);
    tFrom = data[0];
    tTo = data[1];
```

```
    tValue = tFrom + FixMul(tTo - tFrom, tr->percent);
    (tr->dataProc)((struct TweenRecord *)tr, &tValue,
        sizeof(tValue), 1, nil, nil, nil, nil);
    return noErr;
}
```

Resetting a Tween Component

Your tween component must process `kTweenerResetSelect` requests from the Component Manager. Listing 9-3 shows the `TweenReset` function, which resets the component. In this example, because `TweenerInitialize` does not allocate any storage, `TweenerReset` simply returns. In a more complex example, `TweenerReset` releases any storage allocated by `TweenerInitialize` and any storage allocated during the tween operation.

Listing 9-3 Function that resets a tween component

```
pascal ComponentResult TweenerReset (TweenerComponent tc)
{
    return noErr;
}
```

Creating an Interpolation Tween

This section discusses tween operations that modify other tween operations by feeding them artificial time values in place of real time. Listing 9-4 shows how to create an interpolation tween.

Listing 9-4 Creating an interpolation tween container

```
OSErr CreateSampleInterpolatedTweenContainer( QTAtomContainer container,
    TimeValue duration, QTAtom *newTweenAtom )
{
    OSErr           err = noErr;
    Handle          pathData = nil;

    err = QTRemoveChildren( container, kParentAtomIsContainer );
    if ( err )goto bail;

    err = CreateSampleLongTweenContainer( container, 0, duration,
                                        duration, newTweenAtom );
    if ( err ) goto bail;
    pathData = CreateSampleVectorData( 3 );
    if ( ! pathData ) { err = memFullErr; goto bail; }
    err = AddXtoYInterpolatorTweenerForDataSet( container, *newTweenAtom,
                                        *newTweenAtom, 1, pathData );
    if ( err ) goto bail;
bail:
    if ( pathData )DisposeHandle( pathData );
    return err;
}

OSErr AddXtoYInterpolatorTweenerForDataSet( QTAtomContainer container,
```

```
    QTAtom sequenceTweenAtom, QTAtom tweenAtom, QTAtomID dataSetID,
    Handle vectorCodecData )
{
    OSErr               err = noErr;
    QTAtomID            interpolationTweenID;
    QTAtom              dataSetAtom, interpolatorTweenAtom, durationAtom,
                        interpolatorIDAtom;
    TimeValue           duration;
    ComponentInstance   ci = nil;
    UInt8               saveState;
    gxPaths             *thePathData;
    long                dataSize, numPoints;
    gxPoint             firstPoint, lastPoint;
    Boolean             ptIsOnPath;
    Fixed               minOutput, maxOutput;

    if ( (! container) || (! dataSetID) || (! vectorCodecData) )
                                        { err = paramErr; goto bail; }
    saveState = HGetState( vectorCodecData );
    dataSetAtom = QTFindChildByID( container, tweenAtom, kTweenData,
                                dataSetID, nil );
    if ( ! dataSetAtom ) { err = cannotFindAtomErr; goto bail; }

    // determine duration of tweenEntry so we can use the same duration
    // for the interpolator tween
    durationAtom = QTFindChildByIndex( container, tweenAtom,
                                    kTweenDuration, 1, nil );
    if ( ! durationAtom ) { err = cannotFindAtomErr; goto bail; }

    err = QTCopyAtomDataToPtr( container, durationAtom, false,
                                sizeof(duration), &duration, nil );
    if ( err ) goto bail;

    // determine the minOutput and maxOutput values based for the given
    // vector codec data
    err = OpenADefaultComponent( decompressorComponentType,
                                kVectorCodecType, &ci );
    if ( err ) goto bail;

    HLock( vectorCodecData );

    err = CurveGetAtomDataFromVectorStream ( ci, vectorCodecData,
                        kCurvePathAtom, &dataSize, (Ptr *)&thePathData );
    if ( err ) goto bail;
    err = CurveCountPointsInPath( ci, thePathData, 0,
                                (unsigned long *)&numPoints );
    if ( err ) goto bail;
    err = CurveGetPathPoint( ci, thePathData, 0, 0, &firstPoint,
                            &ptIsOnPath );
    if ( err ) goto bail;
    err = CurveGetPathPoint( ci, thePathData, 0, numPoints - 1,
                            &lastPoint, &ptIsOnPath );
    if ( err ) goto bail;
    minOutput = firstPoint.x;
    maxOutput = lastPoint.x;

    // add interolator tween atom with any unique id
    err = AddTweenAtom( container, sequenceTweenAtom, 0,
```

```
                        kTweenTypePathXtoY, 0, duration, minOutput,
                        maxOutput, nil, &interpolatorTweenAtom );
    if ( err ) goto bail;
    // so what was that unique id?
    err = QTGetAtomTypeAndID( container, interpolatorTweenAtom, nil,
                                &interpolationTweenID );
    if ( err ) goto bail;
    err = AddDataAtom( container, interpolatorTweenAtom, 1,
                        GetHandleSize( vectorCodecData ),
                        *vectorCodecData, nil, 0, nil );
    if ( err ) goto bail;

    // finally, we need to reference this new interpolator tween
    interpolatorIDAtom  = QTFindChildByID( container, tweenAtom,
                            kTweenInterpolationID, dataSetID, nil );
    if ( ! interpolatorIDAtom ) {
        err = QTInsertChild( container, tweenAtom, kTweenInterpolationID,
                            dataSetID, 0, 0, nil, &interpolatorIDAtom );
        if ( err ) goto bail;
    }
    err = QTSetAtomData( container, interpolatorIDAtom,
                sizeof(interpolationTweenID), &interpolationTweenID );
    if ( err ) goto bail;
bail:
    if ( vectorCodecData )
        HSetState( vectorCodecData, saveState );
    return err;
}
```

To scale the output of an interpolation tween, you add the optional `kTweenOutputMaxValue` atom and `kTweenOutputMinValue` atom.

Document Revision History

This table describes the changes to *QuickTime Media Types and Media Handlers Guide*.

Date	Notes
2006-01-10	New document that describes media handlers for video, sound, text, time codes, and tweens.
	Replaces "Media Handlers: Introduction, Video and Sound," "Text Media Handlers," "Time Code Media Handlers," "Tween Components and Tween Media" and "Tween Media Handler."
2002-09-17	New document that introduces the QuickTime components for processing video and sound media.