QuickTime 2.0 SDK:

Toolbox Changes

Apple Computer, Inc.

Apple Computer, Inc.

TABLE OF CONTENTS

Table of Contents iii

About this guide vii

Chapter 1 Movie Toolbox 1

Preloading Tracks 1
Hints 1
Data References 2
Track References 2
Timecode Media Handler 2
Data Handler Components 3
Movie Toolbox Reference 3
Functions for Getting and Playing Movies 3
Movie Functions 3
Movie Functions 8
Enhancing Movie Playback Performance 8
Working with Progress and Cover Functions 11
Functions That Modify Movie Properties 12
Working With Movie Spatial Characteristics 12
Locating a Movie's Tracks and Media Structures 14
Working With Track References 15
Functions for Editing Movies 19
Adding Samples to Media Structures 19
Media Functions 21
Selecting Data Handlers 21
Timecode Media Handler Functions 22

Chapter 2 Image Compression Manager 35

Image Compression Manager Reference 35
Image Compression Manager Routines 35
Working With Sequences 35

Chapter 3 Image Compressor Components 41

Image Compressor Components Reference 42
Data Types 42
The Compressor Capability Structure 42
The Decompression Parameters Structure 42
Functions 44
Indirect Functions 44
Image Compression Manager Utility Functions 46

Chapter 4 Sequence Grabber Components 49

Sequence Grabber Components Reference 49
Sequence Grabber Component Functions 49
Configuring Sequence Grabber Components 49
Controlling Sequence Grabber Components 55
Working With Channel Characteristics 55
Working with Sequence Grabber Outputs 57

Chapter 5 Sequence Grabber Channel Components 67
Sequence Grabber Components Reference 67
Sequence Grabber Channel Component Functions 67
Configuration Functions for All Channel Components 67
Chapter 6 Video Digitizer Components 69
Video Digitizer Components Reference 69
Video Digitizer Component Functions 69
Controlling Digitization 69
Utility Functions 71
Chapter 7 Movie Data Exchange Components 73
Direct Importation 73
Audio CD Import Component 73
Movie Data Exchange Components Reference 74
Importing Movie Data 74
Chapter 8 Derived Media Handler Components 75
Derived Media Handler Components Reference 75
Functions 75
Managing Your Media Handler Component 75
Graphics Data Management 76
Base Media Handler Utility Functions 77
Chapter 9 Data Handler Components 79
About Data Handler Components 81
Data Handler Components 81
Using Data Handler Components 84
Selecting a Data Handler 85
Selecting by Component Type Value 85
Interrogating a Data Handler's Capabilities 86
Managing Data References 87
Retrieving Movie Data 87
Storing Movie Data 88
Managing the Data Handler 88
Creating a Data Handler Component 89
General Information 89
Macintosh Data Handler Components 90
Sample Macintosh Data Handler 90
Windows Data Handler Components 100
Sample Windows Data Handler 100
Reference to Data Handler Components 128
Functions 128
Selecting a Data Handler 128
Working With Data References 134
Reading Movie Data 139
Writing Movie Data 146
Managing Data Handler Components 153
Completion Function 155

Chapter 10 QuickTime Music Architecture 157
QuickTime Music Architecture Overview 157
General Terminology 159
Advantages of QuickTime Music Architecture 162
Components of QuickTime Music Architecture 163
Tune Player 163
Note Allocator 164
Music Component 165
Event Sequence Format 168
General Event 170
Note Event 171
Extended Note Event 172
Rest Event 173
End Marker Event 174
Controller Event 175
Extended Controller Event 176
Knob Event 177
Component Interfaces 178
Tune Player 178
Sequence Data 179
Sequence Control 182
Note Allocator 190
Note Channel Allocation and Use 190
Miscellaneous Interface Tools 208
System Configuration 213
Music Component Interface 220
Synthesizer Access 220
Instrument Control 230
Part Access 242
Synthesizer Timing 252
Conversion of Standard MIDI 254
Music Configuration Utility 255
Appendix 257
General MIDI Instrument Numbers 257
General MIDI DrumKit Numbers 259
General MIDI Kit Names 259
Index 261

ABOUT THIS GUIDE

This is the delta guide for QuickTime 2.0 for the Macintosh. This document describes how developers can take advantage of the new features in QuickTime 2.0. Before reading this document, you should be familiar with QuickTime and with the existing QuickTime technical documentation.

This document is organized much like the *Inside Macintosh* books on QuickTime. There are separate chapters for each part of QuickTime—the chapter titles correspond to chapters in the current books. Within these chapters, the section headings also correspond to existing sections wherever possible. In some cases, this document contains new chapters and sections to address completely new areas of functionality.

Briefly, this document contains the following chapters:

- Chapter 1, “The Movie Toolbox,” describes new Movie Toolbox features in QuickTime 2.0, including new support for track references and timecode tracks.
- Chapter 2, “Image Compression Manager,” discusses new Image Compression Manager functionality, especially the new support for scheduled asynchronous decompression operations.
- Chapter 3, “Image Compressor Components,” describes how compressor and decompressor components have changed in order to support the new image-compression features of QuickTime 2.0.
- Chapter 4, “Sequence Grabber Components,” provides information about new sequence-grabber features; in particular, QuickTime 2.0 introduces the concept of a sequence grabber output.
- Chapter 5, “Sequence Grabber Channel Components,” describes how sequence grabber channel components have changed in order to support new sequence-grabber functionality.
- Chapter 6, “Video Digitizer Components,” discusses new video digitizer component features, including support for timecode tracks.
- Chapter 7, “Movie Data Exchange Components,” presents information about new data import and export features of QuickTime 2.0.
- Chapter 8, “Derived Media Handler Components,” discusses changes that affect derived media handler components.
- Chapter 9, “Data Handler Components,” describes the interface that must be supported by QuickTime data handler components. While data handler components have been a part of QuickTime since its inception, this is the first time that Apple has described the interface supported by these components.

- Chapter 10, “QuickTime Music Architecture,” describes QuickTime's new support for music in QuickTime movies. This is an entirely new part of QuickTime. With the QuickTime Music Architecture your application can allow users to play, edit, cut, copy and paste movie music data in the same way they work with text and graphic elements today.

- Appendix, displays the General MIDI Kit Names, DrumKit Numbers, and Instrument Numbers.

CHAPTER 1 MOVIE TOOLBOX

This chapter discusses the changes to the Movie Toolbox. The following sections discuss major new areas of functionality. The reference section provides the details of how to use these new features.

PRELOADING TRACKS

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

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

HINTS

QuickTime 2.0 defines several new movie and media playback hints:

`hintsDontPurge`	Instructs the Movie Toolbox not to dispose of movie data after playing it. The Movie Toolbox leaves the data in memory, in a purgeable handle. This can enhance the playback of small movies that are looping. However, it may consume large amounts of memory and affect the performance of the Memory Manager. Use this hint carefully.
`hintsInactive`	Tells the Movie Toolbox that the movie is not in an active window. This can allow the Movie Toolbox to more efficiently allocate scarce system resources. The movie controller component uses this hint for all movies it manages.

These new hints work with the `SetMoviePlayHints` and `SetMediaPlayHints` functions.

DATA REFERENCES

The Movie Toolbox now fully supports a media that refers to data in more than one file. In the past, a media was restricted to a single data file. By allowing a single media to refer to more than one file, the Movie Toolbox allows better playback performance and easier editing, primarily by reducing the number of tracks in a movie. Use the new `SetMediaDefaultDataRefIndex` function to control which of a media's files you access when you add new samples. See "Adding Samples to Media Structures," later in this chapter, for a complete description of this new function.

Track References

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

Track references can be useful in a variety of ways. In QuickTime 2.0, track references are used to relate timecode tracks to other movie tracks (see "Timecode Media Handler," elsewhere in this chapter, for more information about timecode tracks). You might consider using track references to identify relationships between video and sound tracks, identifying the track that contains dialog and the track that contains background sounds, for example. Another use of track references is to associate one or more text tracks that contain subtitles with the appropriate audio track or tracks.

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

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

See "Working With Track References," later in this chapter, for detailed descriptions of the Movie Toolbox functions that allow you to work with track references.

Timecode Media Handler

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

See “Timecode Media Handler Functions,” later in this chapter, for detailed descriptions of the timecode media handler.

DATA HANDLER COMPONENTS

QuickTime 2.0 includes a new, memory-based data handler. This data handler component works with movie data that is stored in memory, in a handle, rather than in a file. This data handler has a component subtype value of `HandleDataHandlerSubType` (`'hndl'`).

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

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

Although data handler components have been existing since QuickTime 1.0, their interface is publicly defined for the first time in QuickTime 2.0. If you are interested in developing a data handler, refer to the chapter “Data Handler Components” later in this document.

MOVIE TOOLBOX REFERENCE

This section contains reference material on new or changed Movie Toolbox functions.

Functions for Getting and Playing Movies

Movie Functions

NewMovieFromUserProc

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

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

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

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

`newMovieActive` Controls whether the new movie is active. Set this flag to 1 to make the new movie active. You can make a movie active or inactive by calling the `SetMovieActive` function.

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

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

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

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

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

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

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

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

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

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

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

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

DESCRIPTION

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

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

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

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

SPECIAL CONSIDERATIONS

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

ERROR CODES

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

NewMovieFromFile

The `NewMovieFromFile` function now works with some files that do not contain movie resources. In some cases, the data in a file is already sufficiently well-formatted for QuickTime or its components to understand. For example, the AIFF movie data import component can understand AIFF sound files and import the sound data into a QuickTime movie. When the `NewMovieFromFile` function encounters a file that does not contain a movie resource, the function now tries to find a movie import component that can understand the data and create a movie. For more information about new capabilities of movie data import components, see the chapter "Movie Data Exchange Components" elsewhere in this document.

ConvertMovieToFile

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

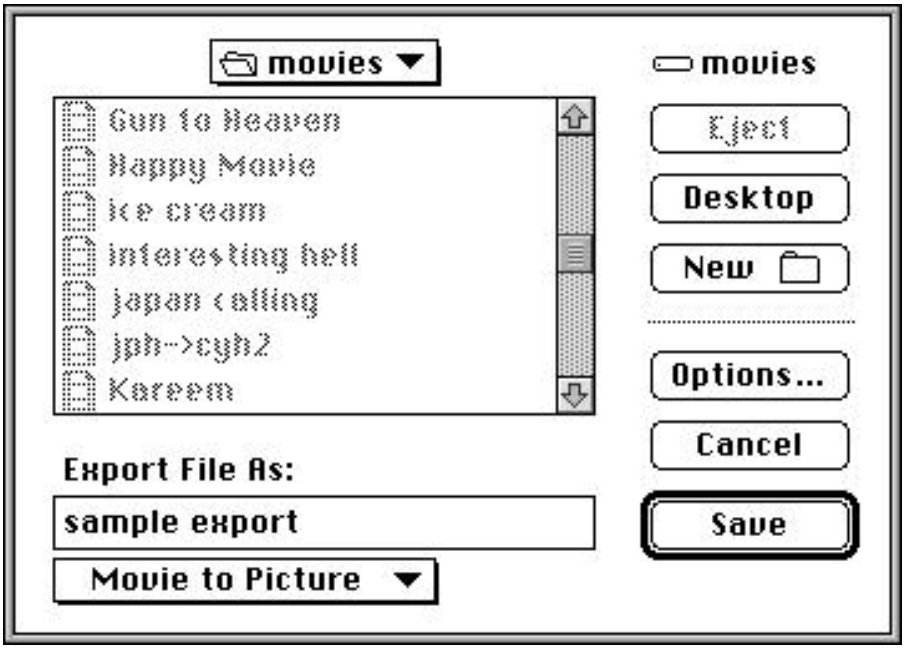


Figure 1 Sample “Save As...” dialog box

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

`showUserSettingsDialog`
Controls whether the “Save As...” dialog can appear. Set this flag to 1 to use the “Save As...” dialog.

`movieToFileOnlyExport`
Restricts the user to export file formats. If you want to require the user to export the movie data using a movie data export component, set this flag to 1. The dialog then displays only file types that are supported by movie data export components.

The following code shows how to call this function.

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

Movie Functions

FlattenMovie and FlattenMovieData

The Movie Toolbox, via the new `SetTrackLoadSettings` function, now allows you to set a movie's preloading guidelines when you create the movie. The preload information is preserved when you flatten the movie (using either the `FlattenMovie` or `FlattenMovieData` functions). In flattened movies, the tracks that are to be preloaded are stored at the start of the movie, rather than being interleaved with the rest of the movie data. This improves preload performance.

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

Enhancing Movie Playback Performance

SetTrackLoadSettings

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

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

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

`preloadTime` Specifies the starting point of the portion of the track to be preloaded. Set this parameter to –1 if you want to preload the entire track (in this case the function ignores the `preloadDuration` parameter).

`preloadDuration` Specifies the amount of the track to be preloaded, starting from the time specified in the `preloadTime` parameter. If you are preloading the entire track, the function ignores this parameter.

`preloadFlags` Controls when the Movie Toolbox preloads the track. The function supports the following flag values:

`preloadAlways` Specifies that the Movie Toolbox should always preload this track, even if the track is disabled.

`preloadOnlyIfEnabled` Specifies that the Movie Toolbox should preload this track only when the track is enabled.

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

`defaultHints` Specifies playback hints for the track. You may specify any of the supported hints flags. See "Hints," earlier in this chapter, for some flags that are new with QuickTime 2.0.

DESCRIPTION

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

SPECIAL CONSIDERATIONS

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

ERROR CODES

`invalidTrack`	–2009	This track is corrupted or invalid

GetTrackLoadSettings

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

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

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

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

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

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

`preloadAlways` Specifies that the Movie Toolbox always preloads this track.

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

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

ERROR CODES

`invalidTrack`	–2009	This track is corrupted or invalid

Working with Progress and Cover Functions

SetMovieDrawingCompleteProc

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

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

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

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

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

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

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

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

DESCRIPTION

Your drawing-complete function must support the following interface:

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

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

ERROR CODES

`invalidMovie`	–2010	Your movie reference is bad

Functions That Modify Movie Properties

Working With Movie Spatial Characteristics

SetMovieColorTable

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

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

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

DESCRIPTION

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

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

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

ERROR CODES

`invalidMovie`	–2010	Your movie reference is bad
Memory Manager errors		

GetMovieColorTable

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

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

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

DESCRIPTION

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

ERROR CODES

`invalidMovie`	–2010	Your movie reference is bad
Memory manager errors		

Locating a Movie's Tracks and Media Structures

GetMovieIndTrackType

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

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

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

`index` Specifies the index value of the track for this operation. This is not that same as the track's index value in the movie. Rather, this parameter is an index into the set of tracks that meet your other selection criteria.

`trackType` Contains either a media type or a media characteristic value. The Movie Toolbox applies this value to the search, and returns information about tracks that meet this criterion. You indicate whether you have specified a media type or characteristic value by setting the `flags` parameter appropriately.

`flags` Contains flags that control the search operation. The following flags are valid (note that you may not set both `movieTrackMediaType` and `movieTrackCharacteristic` to 1):

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

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

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

DESCRIPTION

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

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

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

ERROR CODES

`paramErr`	–50	Invalid parameter specified
`invalidMovie`	–2010	Your movie reference is bad

Working With Track References

Track references allow you to relate tracks to one another. This can be useful for identifying the text track that contains the subtitles for a movie's audio track, and relating the text track to a particular audio track. See "Track References," earlier in this chapter, for more information about track references.

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

AddTrackReference

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

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

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

ERROR CODES

`invalidTrack`	–2009	This track is corrupted or invalid

Memory Manager errors

DeleteTrackReference

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

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

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

DESCRIPTION

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

ERROR CODES

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

Memory Manager errors

SetTrackReference

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

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

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

ERROR CODES

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

GetTrackReference

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

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

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

DESCRIPTION

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

GetNextTrackReferenceType

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

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

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

DESCRIPTION

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

GetTrackReferenceCount

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

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

`theTrack`	Identifies the track for this operation. Your application obtains this track identifier from such Movie Toolbox functions as `NewMovieTrack` and `GetMovieTrack`.
`refType`	Specifies the type of reference. The Movie Toolbox determines the number of track references of this type.

DESCRIPTION

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

Functions for Editing Movies

Adding Samples to Media Structures

SetMediaDefaultDataRefIndex

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

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

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

DESCRIPTION

Since the Movie Toolbox has never allowed you to create tracks that have data in several files, there has not been a mechanism for controlling which data reference is affected by a media editing session. The `SetMediaDefaultDataRefIndex` function allows you to specify the index of the data reference to be edited. After calling this function, you can start editing that data reference by calling the `BeginMediaEdits` function.

ERROR CODES

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

SetMediaPreferredChunkSize

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

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

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

DESCRIPTION

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

ERROR CODES

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

GetMediaPreferredChunkSize

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

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

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

ERROR CODES

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

Media Functions

Selecting Data Handlers

GetDataHandler

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

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

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

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

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

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

`kDataHCanRead` Specifies that you intend to use the data handler component to read data.

`kDataHCanWrite` Specifies that you intend to use the data handler component to write data.

`kDataHCanStreamingWrite` Indicates that you intend to do streaming writes (as part of a movie-capture operation, for example).

DESCRIPTION

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

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

Given that even the most-appropriate data handler component may not support all of the functionality you desire, you should query that component’s capabilities before you start reading or writing movie data.

ERROR CODES

Memory Manager errors

Timecode Media Handler Functions

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

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

The timecode media handler allows QuickTime movies to store additional timing information that is not created by or for QuickTime. This additional timing information would typically be derived from the original source material, say 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.

As with any movie data, a movie’s timecode is stored in a timecode track. Timecode tracks contain

- Source identification information (this identifies the source, say, 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 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 know timecode standards, including, for example, SMPTE timecode. 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 contains 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 skew 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 have the Movie Toolbox display the timecode when a movie is played. Use the `TCSetTimeCodeFlags` function to turn the timecode display on and off. Note that the timecode track must be enabled for this display to work.

You store the timecode's source identification information in a user data item. Create a user data item with a type value of `TCSourceRefNameType` (`'name'`). 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. Be sure to note the index value that you assign to the user data item. You will need it in order to create timecode sample descriptions. For more information about working with user data, see *Inside Macintosh: QuickTime*.

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

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

{
   UserData srcRef;

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

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

```
      }
      DisposeUserData(srcRef);
   }
}
```

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

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

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

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

Field Descriptions

`descSize`	Specifies the size of the sample description, in bytes.
`dataFormat`	Indicates the sample description type (`TimeCodeMediaType`, or `'tmcd'`).
`resvd1`	Reserved for use by Apple. Set this field to 0.

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

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

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

Field Descriptions

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

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

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

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

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

TCGetCurrentTimeCode

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

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

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

ERROR CODES

`invalidTime`	–2015	This time value is invalid

TCGetTimeCodeAtTime

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

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

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

ERROR CODES

`invalidTime`	–2015	This time value is invalid
Memory Manager errors		

TCTimeCodeToFrameNumber

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

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

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

ERROR CODES

`paramErr`	–50	Invalid parameter specified

TCFrameNumberToTimeCode

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

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

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

ERROR CODES

`paramErr`	–50	Invalid parameter specified

TCTimeCodeToString

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

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

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

ERROR CODES

`paramErr`	–50	Invalid parameter specified

TCSetSourceRef

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

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

`mh`	Specifies the timecode media handler. You obtain this identifier by calling the `GetMediaHandler` function.
`tcdH`	Specifies a handle containing the timecode media sample reference that is to be updated.
`srefH`	Specifies a handle to the source information to be placed in the sample reference. It is your application's responsibility to dispose of this handle when you are done with it.

ERROR CODES

`paramErr`	–50	Invalid parameter specified
Memory Manager errors		

TCGetSourceRef

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

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

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

`tcdH` Specifies a handle containing the timecode media sample reference for this operation.

`srefH` Specifies a pointer to a handle that will receive the source information. It is your application's responsibility to dispose of this handle when you are done with it.

ERROR CODES

`paramErr`	–50	Invalid parameter specified
Memory Manager errors		

TCSetTimeCodeFlags

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

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

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

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

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

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

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

TCGetTimeCodeFlags

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

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

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

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

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

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

TCSetDisplayOptions

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

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

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

`textOptions`	Contains a pointer to a text options structure. This structure contains font and style information.

DESCRIPTION

You provide the text style information in a text options structure. This structure is defined as follows (for more information about working with text characteristics, see *Inside Macintosh: Text*):

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

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

TCGetDisplayOptions

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

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

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

ERROR CODES

`paramErr`	–50	Invalid parameter specified

CHAPTER 2 IMAGE COMPRESSION MANAGER

This chapter discusses new features in the Image Compression Manager.

QuickTime 2.0 introduces the concept of scheduled asynchronous decompression operations. Decompressor components can now allow applications to queue decompression operations and specify when those operations should take place. See the chapter “Image Compressor Components” for more information.

The Image Compression Manager provides a new function, `DecompressSequenceFrameWhen`, that allows your application to schedule an asynchronous decompression operation. This function is described later in this chapter.

As discussed in the “Movie Toolbox” chapter of this document, QuickTime 2.0 also introduces timecode tracks to QuickTime movies. Both the Image Compression Manager and compressor components have been enhanced to support timecode information. The Image Compression Manager now provides the `SetDSequenceTimeCode` function, which allows you to set the timecode value for a frame that is to be decompressed. For more information about timecodes and the timecode media handler, see the “Movie Toolbox” chapter earlier in this document.

IMAGE COMPRESSION MANAGER REFERENCE

Image Compression Manager Routines

Working With Sequences

DecompressSequenceFrameWhen

The `DecompressSequenceFrameWhen` function allows you to queue a frame for decompression and specify the time at which the Image Compression Manager is to perform the decompression.

```
pascal OSErr DecompressSequenceFrameWhen (ImageSequence
                                seqID, Ptr data,
                                long dataSize,
                                CodecFlags inFlags,
                                CodecFlags *outFlags,
                                ICMCompletionProcRecordPtr
                                asyncCompletionProc, const
                                ICMFrameTimePtr frameTime);
```

`seqID` Contains the unique sequence identifier that was returned by the `DecompressSequenceBegin` function.

`data` Points to the compressed image data. This pointer must contain a 32-bit clean address. If you use a dereferenced, locked handle, you must call the Memory Manager's `StripAddress` function before you use that pointer with this parameter.

`inFlags` Contains flags providing further control information. See *Inside Macintosh: QuickTime* for information about `CodecFlags` fields. The following flags are valid for this function:

`codecFlagNoScreenUpdate`
Controls whether the decompressor updates the screen image. If you set this flag to 1, the decompressor does not write the current frame to the screen, but does write the frame to its offscreen image buffer (if one was allocated). If you set this flag to 0, the decompressor writes the frame to the screen.

`codecFlagDontOffscreen`
Controls whether the decompressor uses the offscreen buffer during sequence decompression. This flag is only used with sequences that have been temporally compressed. If this flag is set to 1, the decompressor does not use the offscreen buffer during decompression. Instead, the decompressor returns an error. This allows your application to refill the offscreen buffer. If this flag is set to 0, the decompressor uses the offscreen buffer if appropriate.

`codecFlagOnlyScreenUpdate`
: Controls whether the decompressor decompresses the current frame. If you set this flag to 1, the decompressor writes the contents of its offscreen image buffer to the screen, but does decompress the current frame. If you set this flag to 0, the decompressor decompresses the current frame and writes it to the screen. You can set this flag to 1 only if you have allocated an offscreen image buffer for use by the decompressor.

`outFlags`
: Contains status flags. The decompressor updates these flags at the end of the decompression operation. See *Inside Macintosh: QuickTime* for information about `CodecFlags` constants. The following flags may be set by this function:

`codecFlagUsedNewImageBuffer`
: Indicates to your application that the decompressor used the offscreen image buffer for the first time when it processed this frame. If this flag is set to 1, the decompressor used the image buffer for this frame and this is the first time the decompressor used the image buffer in this sequence.

`codecFlagUsedImageBuffer`
: Indicates whether the decompressor used the offscreen image buffer. If the decompressor used the image buffer during the decompress operation, it sets this flag to 1. Otherwise, it sets this flag to 0.

`codecFlagDontUseNewImageBuffer`
: Forces an error to be returned when a new image buffer would have to be allocated instead of allocating the new buffer.

`codecFlagInterlaceUpdate`
: Updates the screen interlacing even and odd scan lines to reduce tearing artifacts (if the decompressor supports this mode).

`asyncCompletionProc`
: Points to a completion function structure. The compressor calls your completion function when an asynchronous decompression operation is complete. You can cause the decompression to be performed asynchronously by specifying a completion function. See *Inside Macintosh: QuickTime* for more information about completion functions.

  If you specify asynchronous operation, you must not read the decompressed image until the decompressor indicates that the operation is complete by calling your completion function. Set `asyncCompletionProc` to `nil` to specify synchronous decompression. If you set `asyncCompletionProc` to –1, the operation is performed asynchronously but the decompressor does not call your completion function.

`frameTime`
: Points to a structure that contains the frame's time information, including the time at which the frame should be displayed, its duration, and the movie's playback rate.

DESCRIPTION

This function accepts the same parameters as the `DecompressSequenceFrame` function, with the addition of the `frameTime` parameter. This parameter points to an `ICMFrameTime` structure, which contains the frame's time information. This structure is discussed in "Image Compressor Components," later in this document.

SPECIAL CONSIDERATIONS

If the current decompressor component does not support this function, the Image Compression Manager returns an error code of `codecCantWhenErr`. If the decompressor cannot service your request at a particular time (say, it's queue is full), the Image Compression Manager returns an error code of `codecCantQueueErr`. The best way to determine whether a decompressor component supports this function is to go ahead and call the function—a component's ability to honor the request may change based on screen depth, clipping settings, and so on.

ERROR CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memFullErr`	–108	Not enough memory available
`noCodecErr`	–8961	Could not find the specified decompressor
`codecSpoolErr`	–8966	Error loading or unloading data
`codecCantWhenErr`	–8974	Decompressor can't honor this request
`codecCantQueueErr`	–8975	Decompressor can't queue this frame

SetDSequenceTimeCode

The `SetDSequenceTimeCode` function allows you to set the timecode value for the frame that is about to be decompressed.

```
pascal OSErr SetDSequenceTimeCode (ImageSequence seqID,
                                   const TimeCodeDef *timeCodeFormat,
                                   const TimeCodeTime *timeCodeTime);
```

`seqID`	Contains the unique sequence identifier that was returned by the `DecompressSequenceBegin` function.
`timeCodeFormat`	Contains a pointer to a timecode definition structure. You provide the appropriate timecode definition information for the next frame to be decompressed.
`timeCodeTime`	Contains a pointer to a timecode record structure. You provide the appropriate time value for the next frame in the current sequence.

DESCRIPTION

QuickTime's video media handler uses this function to set the timecode information for a movie. When a movie that contains timecode information starts playing, the media handler calls this function as it processes the movie's first frame.

Note that the Image Compression Manager passes the timecode information straight through to the image decompressor component. That is, the Image Compression Manager does not make a copy of any of this timecode information. As a result, you must make sure that the data referred to by the `timeCodeFormat` and `timeCodeTime` parameters is valid until the next decompression operation completes.

ERROR CODES

`noErr`	0	No error
`paramErr`	–50	Invalid parameter specified
`memFullErr`	–108	Not enough memory available
`noCodecErr`	–8961	Could not find the specified decompressor

CHAPTER 3 IMAGE COMPRESSOR COMPONENTS

In QuickTime 2.0 the Image Compression Manager has been enhanced to support scheduled asynchronous decompression operations. By calling the new `DecompressSequenceFrameWhen` Image Compression Manager function, applications can schedule decompression requests in advance. This allows decompressor components that also support this functionality to provide reliable playback performance under a wider range of conditions.

Apple has modified its Cinepak, Video, Animation, Component Video, and Graphics decompressors to support scheduled asynchronous decompression to 8-, 16-, and 32-bit destinations (the Cinepak decompressor also supports 4-bit grayscale destinations).

If you want to support this functionality, you must modify your decompressor component in the following ways:

- Report your component's new capabilities in its compressor capability structure (there are two new flags)
- Modify your component's `CDBandDecompress` function to accept scheduled asynchronous decompression requests and process them correctly
- Implement the new `CDCodecFlush` function; this function allows the Image Compression Manager to instruct you to empty your input queue
- Optionally, implement logic to manage the cursor during decompression operations

All of these changes are discussed in detail in the reference section that follows.

As discussed in the "Movie Toolbox" chapter of this document, QuickTime 2.0 also introduces timecode tracks to QuickTime movies. Both the Image Compression Manager and compressor components have been enhanced to support timecode information. Image compressor components may now support the `CDCodecSetTimeCode` function, which allows the Image Compression Manager to set the timecode value for a frame that is to be decompressed. For more information about timecodes and the timecode media handler, see the "Movie Toolbox" chapter earlier in this document.

IMAGE COMPRESSOR COMPONENTS REFERENCE

Data Types

The Compressor Capability Structure

There are two new decompressor capability flags (your component sets these flags in the `flags` field of the compressor capability structure [`CodecCapabilities`]):

`codecCanAsyncWhen` Indicates whether your decompressor component supports scheduled asynchronous decompression. Set this flag to 1 if your component can support the scheduled variant of the `CDBandDecompress` function. Note that you must also set the `codecCanAsync` flag to 1.

`codecCanShieldCursor`
Indicates whether your decompressor component can shield the cursor during decompression. If your component can manage the cursor's display, set this flag to 1. Your component can use the Image Compression Manager's `ICMShieldSequenceCursor` function to manage the cursor. This function is described later in this chapter in "Image Compression Manager Utility Functions."

Otherwise, set this flag to 0—the Image Compression Manager then manages the cursor for you.

The Decompression Parameters Structure

Apple has modified the definition of the decompression parameters structure. The `frameTime` field has been added. This field contains a pointer to an `ICMFrameTime` structure. This structure contains a frame's time information for scheduled asynchronous decompression operations.

The decompression parameters structure is now defined as follows (the `frameTime` field is near the bottom):

```
typedef struct {
   ImageSequence           sequenceID;       /* unique sequence ID
                                                (predecompress,
                                                banddecompress) */
   ImageDescriptionHandle  imageDescription; /* handle to image
description
                                                structure
(predecompress,
```

```
                                                banddecompress) */
   Ptr                     data;              /* compressed image data */
   long                    bufferSize;        /* size of data buffer */
   long                    frameNumber;       /* frame identifier */
   long                    startLine;         /* starting line for band
*/
   long                    stopLine;          /* ending line for band */
   long                    conditionFlags;    /* condition flags */
   CodecFlags              callerFlags;       /* control flags */
   CodecCapabilitiesPtr    *capabilities;     /* pointer to compressor
                                                capability structure
                                                (predecompress,
                                                banddecompress) */
   ProgressProcRecord      progressProcRecord;
                                              /* progress function
                                                structure */
   CompletionProcRecord    completionProcRecord;
                                              /* completion function
                                                structure */
   DataProcRecord          dataProcRecord;    /* data-loading function
                                                structure */
   CGrafPtr                port;              /* pointer to color
                                                graphics port for image
                                                (predecompress,
                                                banddecompress) */
   PixMap                  dstPixMap;         /* destination pixel map
                                                (predecompress,
                                                banddecompress) */
   BitMapPtr               maskBits;          /* update mask */
   PixMapPtr               mattePixMap;       /* blend matte pixel map */
   Rect                    srcRect;           /* source rectangle
                                                (predecompress,
                                                banddecompress) */
   MatrixRecordPtr         *matrix;           /* pointer to matrix }
                                                structure
                                                (predecompress,
                                                banddecompress) */
   CodecQ                  accuracy;          /* desired accuracy
                                                (predecompress,
                                                banddecompress */
   short                   transferMode;      /* transfer mode
                                                (predecompress,
                                                banddecompress) */
   ICMFrameTimePtr         frameTime          /* time information
                                                (scheduled decompress)
*/
   long                    reserved[1];       /* reserved */

} CodecDecompressParams;
```

The new field is used as follows:

`frameTime`	Contains a pointer to an `ICMFrameTime` structure. This structure contains time information relating to scheduled asynchronous decompression operations.

The `ICMFrameTime` structure is defined as follows:

```
struct ICMFrameTimeRecord {
   Int64Bit                value;          /* time to display frame */
   long                    scale;          /* time scale */
   void                    *base;          /* reference to time base
*/
   long                    duration;       /* display duration */
   Fixed                   rate;           /* movie's playback rate */
};
```

The structure's fields are defined as follows:

`value`	Specifies the time at which the frame is to be displayed. The `scale` field specifies the units for this value; the `base` field refers to the time base.
`scale`	Indicates the units for the frame's display time.
`base`	Refers to the time base.
`duration`	Specifies the duration for which the frame is to be displayed.
`rate`	Indicates the time base effective rate.

Functions

Indirect Functions

CDPreDecompress

If your decompressor component supports scheduled asynchronous decompression operations, be sure to set the `codecCanAsyncWhen` flag to 1 in the `flags` field of your component's compressor capabilities structure.

CDBandDecompress

For scheduled asynchronous decompression operations, the Image Compression Manager supplies a reference to an `ICMFrameTime` structure in this function's decompression parameters structure parameter. The `ICMFrameTime` structure contains time information governing the scheduled decompression operation, including the time at which the frame must be displayed. See "The Decompression Parameters Structure," earlier in this chapter, for a complete description of this structure.

When your component has finished the decompression operation, it must call the application's completion function. In the past, your component called that function directly. For scheduled asynchronous decompression operations, your component should call the Image Compression Manager's `ICMDecompressComplete` function, which is described later in this chapter.

If your component does not support scheduled asynchronous decompression, return an error code of `codecCantWhenErr`. If your component's queue is full, return an error code of `codecCantQueueErr`.

For other asynchronous decompression operations, the Image Compression Manager sets the `frameTime` field in the decompression parameters structure to `nil`.

CDCodecFlush

Your component receives the `CDCodecFlush` function whenever the Image Compression Manager needs to empty your component's input queue.

```
pascal ComponentResult CDCodecFlush;
```

DESCRIPTION

Your component should empty its queue of scheduled asynchronous decompression requests. For each request, your component must call the `ICMDecompressComplete` function. Be sure to set the `err` parameter to –1, indicating that the request was canceled. Also, you must set both the `codecCompletionSource` and `codecCompletionDest` flags to 1.

SPECIAL CONSIDERATIONS

Your component's `CDCodecFlush` function may be called at interrupt time.

CDCodecSetTimeCode

Your component receives `CDCodecSetTimeCode` function whenever an application calls the Image Compression Manager's `SetDSequenceTimeCode` function. That function allows an application to set the timecode for a frame that is to be decompressed.

```
pascal OSErr CDCodecSetTimeCode (ImageSequence seqID,
                            const TimeCodeDef *timeCodeFormat,
                            const TimeCodeTime *timeCodeTime);
```

`seqID`	Contains the unique sequence identifier that was returned by the `DecompressSequenceBegin` function.
`timeCodeFormat`	Contains a pointer to a timecode definition structure. This structure contains the timecode definition information for the next frame to be decompressed.
`timeCodeTime`	Contains a pointer to a timecode record structure. This structure contains the time value for the next frame in the current sequence.

DESCRIPTION

The timecode information you receive applies to the next frame to be decompressed.

Image Compression Manager Utility Functions

ICMDecompressComplete

Your component must call the `ICMDecompressComplete` function whenever it finishes a scheduled asynchronous decompression operation.

```
pascal void ICMDecompressComplete (ImageSequence seqID,
                                   OSErr err, short flag,
                                   ICMCompletionProcRecordPtr
                                   completionRtn);
```

`seqID`	Identifies the frame's sequence.

`err` Indicates whether the operation succeeded or failed. Set this parameter to 0 for successful operations. For failed operations, set a reasonable result code. For canceled operations (for example, when the Image Compression Manager calls your component's `CDCodecFlush` function), set this parameter to –1.

`flag` Indicates which part of the operation is complete. The following flags are defined:

`codecCompletionSource`
Your component is done with the source buffer. Set this flag to 1 when you are done with the processing associated with the source buffer.

`codecCompletionDest`
Your component is done with the destination buffer. Set this flag to 1 when you are done with the processing associated with the destination buffer.

Note that you may set more than one of these flags to 1.

`completionRtn` Contains a pointer to a completion function structure. That structure identifies the application's completion function, and contains a reference constant associated with the frame.

Your component obtains the completion function structure as part of the decompression parameters structure provided by the Image Compression Manager at the start of the decompression operation.

DESCRIPTION

Your component must call this function at the end of scheduled asynchronous decompression operations. For other types of decompression operations, you may still call the application's completion function directly.

ICMShieldSequenceCursor

Your component may call the `ICMShieldSequenceCursor` function to manage the display of the cursor during decompression operations.

```
pascal OSErr ICMShieldSequenceCursor (ImageSequence seqID);
```

`seqID` Identifies the current sequence.

DESCRIPTION

For correct image display behavior, the cursor must be shielded (hidden) during decompression. By default, the Image Compression Manager handles the cursor for you, hiding it at the beginning of a decompression operation and revealing it at the end.

With the advent of scheduled asynchronous decompression, however, the Image Compression Manager cannot do as precise a job of managing the cursor, because it does not when scheduled operations actually begin and end. While the Image Compression Manager can still manage the cursor, it must hide the cursor when each request is queued, rather than when the request is serviced. This may result in the cursor remaining hidden for long periods of time.

In order to achieve better cursor behavior, you can choose to manage the cursor in your decompressor component. If you so choose, you can use the `ICMShieldSequenceCursor` function to hide the cursor—the Image Compression Manager displays the cursor when you call the `ICMDecompressComplete` function. In this manner, the cursor is hidden only when your component is decompressing and displaying the frame.

SPECIAL CONSIDERATIONS

This function is interrupt-safe.

CHAPTER 4 SEQUENCE GRABBER COMPONENTS

This chapter discusses new features of sequence grabber components.

The sequence grabber now allows you to assign a specific file to each channel. This allows you to collect data into more than one file at a time. This can result in improved performance by defining the files for different channels on different devices. These destination containers are referred to as *sequence grabber outputs.* See "Working with Sequence Grabber Outputs," later in this chapter, for a complete discussion.

The sequence grabber now uses data handler components when writing movie data. This provides greater flexibility, especially when working with special storage devices (such as networks).

As discussed in the "Movie Toolbox" chapter of this document, QuickTime 2.0 introduces timecode tracks to QuickTime movies. The sequence grabber automatically creates timecode tracks if the source video data contains timecode information. In order to support timecode tracks, the sequence grabber also provides two functions that let you identify the source information associated with video data that contains timecode information. For more information about timecodes and the timecode media handler, see the "Movie Toolbox" chapter earlier in this document.

SEQUENCE GRABBER COMPONENTS REFERENCE

Sequence Grabber Component Functions

Configuring Sequence Grabber Components

SGSetDataRef

The `SGSetDataRef` function allows you to specify the destination for a record operation using a data reference, and to specify other options that govern the operation. This function is similar to the `SGSetDataOutput` function, and provides you an alternative way to specify the destination.

```
pascal ComponentResult SGSetDataRef (SeqGrabComponent s,
                                     Handle dataRef,
                                     OSType dataRefType,
                                     long whereFlags);
```

`s` Specifies the component instance that identifies your connection to the sequence grabber component. You obtain this value from the Component Manager's `OpenDefaultComponent` or `OpenComponent` function.

`dataRef` Contains a handle to the information that identifies the destination container.

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

`whereFlags` Contains flags that control the record operation. You must set either the `seqGrabToDisk` flag or the `seqGrabToMemory` flag to 1 (set unused flags to 0):

`seqGrabToDisk` Instructs the sequence grabber component to write the recorded data to a QuickTime movie in the container specified by the `dataRef` parameter. If you set this flag to 1, the sequence grabber writes the data to the container as the data is recorded. Set this flag to 0 if you set the `seqGrabToMemory` flag to 1 (only one of these two flags may be set to 1).

`seqGrabToMemory` Instructs the sequence grabber component to store the recorded data in memory until the recording process is complete. The sequence grabber then writes the recorded data to the container specified by the `dataRef` parameter. This technique provides better performance than recording directly to the container, but limits the amount of data you can record. Set this flag to 1 to record to memory. Set this flag to 0 if you set the `seqGrabToDisk` flag to 1 (only one of these two flags may be set to 1).

`seqGrabDontUseTempMemory`
: Prevents the sequence grabber component from using temporary memory during the record operation. By default, the sequence grabber component and its channel components use as much temporary memory as necessary to perform the record operation. Set this flag to 1 to prevent the sequence grabber component and its channel components from using temporary memory.

`seqGrabAppendToFile`
: Directs the sequence grabber component to add the recorded data to the data fork of the container specified by the `dataRef` parameter. By default, the sequence grabber component deletes the container and creates a new file containing one movie and the corresponding movie resource. Set this flag to 1 to cause the sequence grabber component to append the recorded data to the data fork of the container and create a new movie resource in that file.

`seqGrabDontAddMovieResource`
: Prevents the sequence grabber component from adding the new movie resource to the container specified by the `dataRef` parameter. By default, the sequence grabber component creates a new movie resource and adds that resource to the container. Set this flag to 1 to prevent the sequence grabber component from adding the movie resource to the container. You are then responsible for adding the resource to a file, if you so desire.

`seqGrabDontMakeMovie`
: Prevents the sequence grabber component from creating a movie. By default, the sequence grabber component creates a new movie resource and adds the captured data to that movie. If you set this flag to 1, the sequence grabber still calls your data function, but does not write any data to the movie file.

`seqGrabDataProcIsInterruptSafe`
: Specifies that your data function is interrupt-safe, and may be called at interrupt time. This allows the sequence grabber component to present the captured data as soon as possible. Note that not all sequence grabber channel components may use this feature.

DESCRIPTION

If you are performing a preview operation, you do not need to use the `SGSetDataRef` function.

ERROR CODES

`notEnoughMemoryToGrab`	–9403	Insufficient memory for operation
`notEnoughDiskSpaceToGrab`	–9404	Insufficient disk space for operation

File Manager errors
Memory Manager errors

SGGetDataRef

The `SGGetDataRef` function allows you to determine the data reference that is currently assigned to a sequence grabber component and the control flags that would govern a record operation.

```
pascal ComponentResult SGGetDataRef (SeqGrabComponent s,
                                     Handle *dataRef,
                                     OSType *dataRefType,
                                     long *whereFlags);
```

`s`
: Specifies the component instance that identifies your connection to the sequence grabber component. You obtain this value from the Component Manager's `OpenDefaultComponent` or `OpenComponent` function.

`dataRef`
: Contains a pointer to a handle that is to receive the information that identifies the destination container.

`dataRefType`
: Specifies a pointer to a field that is to receive the type of data reference.

`whereFlags` Contains a pointer to a long integer that is to receive flags that control the record operation. The following flags are defined (unused flags are set to 0):

`seqGrabToDisk` Instructs the sequence grabber component to write the recorded data to a QuickTime movie in the container specified by the `dataRef` parameter. If this flag is set to 1, the sequence grabber writes the data to the container as the data is recorded.

`seqGrabToMemory` Instructs the sequence grabber component to store the recorded data in memory until the recording process is complete. The sequence grabber then writes the recorded data to the container specified by the `dataRef` parameter. This technique provides better performance than recording directly to the movie file, but limits the amount of data you can record. If this flag is set to 1, the sequence grabber component is recording to memory.

`seqGrabDontUseTempMemory` Prevents the sequence grabber component from using temporary memory during the record operation. By default, the sequence grabber component and its channel components use as much temporary memory as necessary to perform the record operation. If this flag is set to 1, the sequence grabber component and its channel components do not use temporary memory.

`seqGrabAppendToFile`
: Directs the sequence grabber component to add the recorded data to the data fork of the container specified by the `dataRef` parameter. By default, the sequence grabber component deletes the container and creates a new file containing one movie and its movie resource. If this flag is set to 1, the sequence grabber component appends the recorded data to the data fork of the container and creates a new movie resource in that file.

`seqGrabDontAddMovieResource`
: Prevents the sequence grabber component from adding the new movie resource to the container specified by the `dataRef` parameter. By default, the sequence grabber component creates a new movie resource and adds that resource to the container. If this flag is set to 1, the sequence grabber component does not add the movie resource to the container. You are then responsible for adding the resource to a file, if you so desire.

`seqGrabDontMakeMovie`
: Prevents the sequence grabber component from creating a movie. By default, the sequence grabber component creates a new movie resource and adds the captured data to that movie. If this flag is set to 1, the sequence grabber still calls your data function, but does not write any data to the container.

`seqGrabDataProcIsInterruptSafe`
: Specifies that your data function is interrupt-safe, and may be called at interrupt time. This allows the sequence grabber component to present the captured data as soon as possible. Note that not all sequence grabber channel components may use this feature.

DESCRIPTION

You set these characteristics by calling the `SGSetDataRef` function, which is described in the previous section. If you have not set these characteristics before calling the `SGGetDataRef` function, the returned data is meaningless.

ERROR CODES

Memory Manager errors

Controlling Sequence Grabber Components

SGGetMode

The `SGGetMode` function provides a convenient mechanism for determining whether a sequence grabber component is in preview mode or record mode.

```
pascal ComponentResult SGGetMode (SeqGrabComponent s,
                                  Boolean *previewMode,
                                  Boolean *recordMode);
```

`s`	Specifies the component instance that identifies your connection to the sequence grabber component. You obtain this value from the Component Manager's `OpenDefaultComponent` or `OpenComponent` function.
`previewMode`	Contains a pointer to a Boolean. The sequence grabber component sets this field to `true` if the component is in preview mode.
`recordMode`	Contains a pointer to a Boolean. The sequence grabber component sets this field to `true` if the component is in record mode.

Working With Channel Characteristics

The sequence grabber now supports two new functions, `SGChannelSetDataSourceName` and `SGChannelGetDataSourceName`, that allow you to work with the source identification information associated with a timecode media. For more information about timecodes and the timecode media handler, see the "Movie Toolbox" chapter earlier in this document.

SGChannelSetDataSourceName

The `SGChannelSetDataSourceName` function allows you to set the source information relating to the timecode data created for a track. You must set this information before you start digitizing.

```
pascal ComponentResult SGChannelSetDataSourceName
                                (SGChannel c,
                                const Str255 name,
                                ScriptCode scriptTag);
```

`c`	Specifies the reference that identifies the channel for this operation. This must be a video channel.
`name`	Identifies a string that contains the source identification information.
`scriptTag`	Specifies the language of the source identification information.

DESCRIPTION

This source information identifies the source of the video data (say, a videotape name). The sequence grabber stores this information with the track's timecode information. If the source does not contain timecode information, or the digitizer does not provide the information, the sequence grabber does not save this information.

This function is supported only by video channels.

SGChannelGetDataSourceName

The `SGChannelGetDataSourceName` function allows you to get the source information relating to the timecode data created for a track.

```
pascal ComponentResult SGChannelGetDataSourceName
                                (SGChannel c, Str255 name,
                                ScriptCode *scriptTag);
```

`c`	Specifies the reference that identifies the channel for this operation. This must be a video channel.
`name`	Identifies a string that is to receive the source identification information. Set this parameter to `nil` if you do not want to retrieve the name
`scriptTag`	Specifies a field that is to receive the source information's language code. Set this parameter to `nil` if you do not want this information.

Working with Sequence Grabber Outputs

In order to allow sequence grabber components to capture to more than one data reference at a time, QuickTime 2.0 introduces the concept of a sequence grabber output. A *sequence grabber output* ties a sequence grabber channel to a specified data reference.

If you are capturing to a single movie file, you can continue to use the `SGSetDataOutput` function (or the new `SGSetDataRef` function) to specify the sequence grabber's destination. However, if you want to capture movie data into several different files or data references, you must use sequence grabber outputs to do so. Even if you are using outputs, you must still use the `SGSetDataOutput` function or the `SGSetDataRef` function to identify where the sequence grabber should create the movie resource.

You are responsible for creating outputs, assigning them to sequence grabber channels, and disposing of them when you are done. Sequence grabber components provide a number of functions for managing outputs: the `SGNewOutput` function creates a new output; the `SGDisposeOutput` function disposes of an output; the `SGSetOutputFlags` function configures the output; the `SGSetChannelOutput` function assigns an output to a channel; and the `SGGetDataOutputStorageSpaceRemaining` function determines how much space is left in the output.

SGNewOutput

The `SGNewOutput` function creates a new sequence grabber output. You specify the output's destination container using a data reference.

```
pascal ComponentResult SGNewOutput (SeqGrabComponent s,
                                    Handle dataRef,
                                    OSType dataRefType,
                                    long whereFlags,
                                    SGOutput *output);
```

`s`	Specifies the component instance that identifies your connection to the sequence grabber component. You obtain this value from the Component Manager's `OpenDefaultComponent` or `OpenComponent` function.
`dataRef`	Contains a handle to the information that identifies the destination container.
`dataRefType`	Specifies the type of data reference. If the data reference is an alias, you must set the parameter to `rAliasType` (`'alis'`), indicating that the reference is an alias.
`whereFlags`	Contains flags that control the record operation. You must set either the `seqGrabToDisk` flag or the `seqGrabToMemory` flag to 1 (set unused flags to 0):

`seqGrabToDisk` Instructs the sequence grabber component to write the recorded data to a QuickTime movie in the container specified by the `dataRef` parameter. If you set this flag to 1, the sequence grabber writes the data to the container as the data is recorded. Set this flag to 0 if you set the `seqGrabToMemory` flag to 1 (only one of these two flags may be set to 1).

`seqGrabToMemory` Instructs the sequence grabber component to store the recorded data in memory until the recording process is complete. The sequence grabber then writes the recorded data to the container specified by the `dataRef` parameter. This technique provides better performance than recording directly to the container, but limits the amount of data you can record. Set this flag to 1 to record to memory. Set this flag to 0 if you set the `seqGrabToDisk` flag to 1 (only one of these two flags may be set to 1).

`seqGrabDontUseTempMemory` Prevents the sequence grabber component from using temporary memory during the record operation. By default, the sequence grabber component and its channel components use as much temporary memory as necessary to perform the record operation. Set this flag to 1 to prevent the sequence grabber component and its channel components from using temporary memory.

`seqGrabAppendToFile`
: Directs the sequence grabber component to add the recorded data to the data fork of the container specified by the `dataRef` parameter. By default, the sequence grabber component deletes the container and creates a new file containing one movie and the corresponding movie resource. Set this flag to 1 to cause the sequence grabber component to append the recorded data to the data fork of the container and create a new movie resource in that file.

`seqGrabDontAddMovieResource`
: Prevents the sequence grabber component from adding the new movie resource to the container specified by the `dataRef` parameter. By default, the sequence grabber component creates a new movie resource and adds that resource to the container. Set this flag to 1 to prevent the sequence grabber component from adding the movie resource to the container. You are then responsible for adding the resource to a file, if you so desire.

`seqGrabDontMakeMovie`
: Prevents the sequence grabber component from creating a movie. By default, the sequence grabber component creates a new movie resource and adds the captured data to that movie. If you set this flag to 1, the sequence grabber still calls your data function, but does not write any data to the movie file.

`seqGrabDataProcIsInterruptSafe`
: Specifies that your data function is interrupt-safe, and may be called at interrupt time. This allows the sequence grabber component to present the captured data as soon as possible. Note that not all sequence grabber channel components may use this feature.

`output`	Contains a pointer to a sequence grabber output. The sequence grabber component returns an output identifier. You can then use this identifier with other sequence grabber component functions.

DESCRIPTION

Once you have created the sequence grabber output, you can use the `SGSetChannelOutput` function to assign the output to a sequence grabber channel.

ERROR CODES

`paramErr`	–50	Invalid parameter specified
File Manager errors		
Memory Manager errors		

SGDisposeOutput

The `SGDisposeOutput` function disposes of an existing output.

```
pascal ComponentResult SGDisposeOutput (SeqGrabComponent s,
                                        SGOutput output);
```

`s`	Specifies the component instance that identifies your connection to the sequence grabber component. You obtain this value from the Component Manager's `OpenDefaultComponent` or `OpenComponent` function.
`output`	Identifies the sequence grabber output for this operation. You obtain this identifier by calling the `SGNewOutput` function.

DESCRIPTION

If any sequence grabber channels are using this output, the sequence grabber component assigns them to an undefined output (and any data captured subsequently is lost until you assign a new output to the channel).

Note that you cannot dispose of an output when the sequence grabber component is in record mode.

ERROR CODES

`cantDoThatInCurrentMode`	–9402	Request invalid in current mode

SGSetChannelOutput

The `SGSetChannelOutput` function allows you to assign an output to a channel.

```
pascal ComponentResult SGSetChannelOutput (SeqGrabComponent
                                s, SGChannel c,
                                SGOutput output);
```

`s`	Specifies the component instance that identifies your connection to the sequence grabber component. You obtain this value from the Component Manager's `OpenDefaultComponent` or `OpenComponent` function.
`c`	Identifies the channel for this operation. Provide your connection identifier. You connect to a channel component by calling the `SGNewChannel` or `SGNewChannelFromComponent` functions.
`output`	Identifies the sequence grabber output for this operation. You obtain this identifier by calling the `SGNewOutput` function.

DESCRIPTION

Note that when you call the `SGSetDataRef` or `SGSetDataOutput` functions the sequence grabber component sets every channel to the specified file or container. If you want to use different outputs, you must use this function to assign the channels appropriately.

One output may be assigned to one or more channels.

ERROR CODES

`badSGChannel`	–9406	Invalid channel specified

SGSetOutputFlags

The `SGSetOutputFlags` function allows you to configure an existing sequence grabber output.

```
pascal ComponentResult SGSetOutputFlags (SeqGrabComponent s,
                                         SGOutput output,
                                         long whereFlags);
```

`s` Specifies the component instance that identifies your connection to the sequence grabber component. You obtain this value from the Component Manager's `OpenDefaultComponent` or `OpenComponent` function.

`output` Identifies the sequence grabber output for this operation. You obtain this identifier by calling the `SGNewOutput` function.

`whereFlags` Contains flags that control the record operation. You must set either the `seqGrabToDisk` flag or the `seqGrabToMemory` flag to 1 (set unused flags to 0):

`seqGrabToDisk` Instructs the sequence grabber component to write the recorded data to a QuickTime movie in the container specified by the `dataRef` parameter. If you set this flag to 1, the sequence grabber writes the data to the container as the data is recorded. Set this flag to 0 if you set the `seqGrabToMemory` flag to 1 (only one of these two flags may be set to 1).

`seqGrabToMemory` Instructs the sequence grabber component to store the recorded data in memory until the recording process is complete. The sequence grabber then writes the recorded data to the container specified by the `dataRef` parameter. This technique provides better performance than recording directly to the container, but limits the amount of data you can record. Set this flag to 1 to record to memory. Set this flag to 0 if you set the `seqGrabToDisk` flag to 1 (only one of these two flags may be set to 1).

`seqGrabDontUseTempMemory`
: Prevents the sequence grabber component from using temporary memory during the record operation. By default, the sequence grabber component and its channel components use as much temporary memory as necessary to perform the record operation. Set this flag to 1 to prevent the sequence grabber component and its channel components from using temporary memory.

`seqGrabAppendToFile`
: Directs the sequence grabber component to add the recorded data to the data fork of the container specified by the `dataRef` parameter. By default, the sequence grabber component deletes the container and creates a new file containing one movie and the corresponding movie resource. Set this flag to 1 to cause the sequence grabber component to append the recorded data to the data fork of the container and create a new movie resource in that file.

`seqGrabDontAddMovieResource`
: Prevents the sequence grabber component from adding the new movie resource to the container specified by the `dataRef` parameter. By default, the sequence grabber component creates a new movie resource and adds that resource to the container. Set this flag to 1 to prevent the sequence grabber component from adding the movie resource to the container. You are then responsible for adding the resource to a file, if you so desire.

`seqGrabDontMakeMovie`
: Prevents the sequence grabber component from creating a movie. By default, the sequence grabber component creates a new movie resource and adds the captured data to that movie. If you set this flag to 1, the sequence grabber still calls your data function, but does not write any data to the movie file.

`seqGrabDataProcIsInterruptSafe`
: Specifies that your data function is interrupt-safe, and may be called at interrupt time. This allows the sequence grabber component to present the captured data as soon as possible. Note that not all sequence grabber channel components may use this feature.

ERROR CODES

`paramErr`	–50	Invalid parameter specified
`cantDoThatInCurrentMode`	–9402	Request invalid in current mode

SGGetDataOutputStorageSpaceRemaining

The `SGGetDataOutputStorageSpaceRemaining` function, besides having the longest name in captivity, allows you to determine the amount of space remaining in the file associated with an output.

```
pascal ComponentResult SGGetDataOutputStorageSpaceRemaining
                                (SeqGrabComponent s,
                                SGOutput output,
                                unsigned long *space);
```

`s`	Specifies the component instance that identifies your connection to the sequence grabber component. You obtain this value from the Component Manager's `OpenDefaultComponent` or `OpenComponent` function.
`output`	Identifies the sequence grabber output for this operation. You obtain this identifier by calling the `SGNewOutput` function.
`space`	Contains a pointer to an unsigned long. The sequence grabber component returns a value that indicates the number of bytes of space remaining in the file associated with the output.

DESCRIPTION

Use this function in place of the `SGGetStorageSpaceRemaining` function in cases where you are working with more than one output.

ERROR CODES

`paramErr`	–50	Invalid parameter specified

CHAPTER 5 SEQUENCE GRABBER CHANNEL COMPONENTS

This chapter discusses the changes to sequence grabber channel components.

There are a couple of new functions that allow the sequence grabber to request that your channel component observe a specified data rate.

SEQUENCE GRABBER COMPONENTS REFERENCE

Sequence Grabber Channel Component Functions

Configuration Functions for All Channel Components

SGChannelSetRequestedDataRate

The `SGChannelSetRequestedDataRate` function allows the sequence grabber component to specify the maximum rate at which it would like to receive data from your channel component.

```
pascal ComponentResult SGChannelSetRequestedDataRate
                                (SGChannel c,
                                long bytesPerSecond);
```

`c`	Identifies the channel connection for this operation.
`bytesPerSecond`	Specifies the maximum data rate requested by the sequence grabber component. The sequence grabber component sets this parameter to 0 to remove any data-rate restrictions.

DESCRIPTION

The data rate supplied by the sequence grabber component represents a requested data rate. Your component may not be able to observe that rate under all conditions. The sequence grabber component can accommodate your component occasionally exceeding this suggested rate.

ERROR CODES

`badComponentSelector`	0x80008002	Function not supported

SGChannelGetRequestedDataRate

The SGChannelGetRequestedDataRate function allows the sequence grabber component to retrieve the current maximum data rate value from your channel component.

```
pascal ComponentResult SGChannelGetRequestedDataRate
                                (SGChannel c,
                                long *bytesPerSecond);
```

`c`	Identifies the channel connection for this operation.
`bytesPerSecond`	Points to a field that is to receive the maximum data rate requested by the sequence grabber component. Set this field to 0 if the sequence grabber has not set any restrictions.

ERROR CODES

`badComponentSelector`	0x80008002	Function not supported

CHAPTER 6 VIDEO DIGITIZER COMPONENTS

This chapter discusses changes to video digitizer components.

There is a new function, `VDSetDataRate`, that instructs your video digitizer component to observe a specified rate of data delivery.

As discussed in the "Movie Toolbox" chapter of this document, QuickTime 2.0 introduces timecode tracks to QuickTime movies. Video digitizers may return timecode information for an incoming video signal by responding to the new `VDGetTimeCode` function described in this chapter. For more information about timecodes and the timecode media handler, see the "Movie Toolbox" chapter earlier in this document.

VIDEO DIGITIZER COMPONENTS REFERENCE

Video Digitizer Component Functions

Controlling Digitization

VDSetDataRate

The `VDSetDataRate` function instructs your video digitizer component to limit the rate at which it delivers compressed, digitized video data.

```
pascal VideoDigitizerError VDSetDataRate
                              (VideoDigitizerComponent ci,
                              long bytesPerSecond);
```

`ci`	Specifies the video digitizer component for the request. Applications contains this reference from the Component Manager's `OpenComponent` function.
`bytesPerSecond`	Specifies the maximum data rate requested by the application. Applications set this parameter to 0 to remove any data-rate restrictions.

DESCRIPTION

This function is valid only for video digitizer components that can deliver compressed video (that is, components that support the `VDCompressOneFrameAsync` function). Components that support data-rate limiting set the `codecInfoDoesRateConstrain` flag to 1 in the `compressFlags` field of the `VDCompressionList` structure returned by the component in response to the `VDGetCompressionTypes` function.

Your video digitizer component should return this data-rate limit in the `bytesPerSecond` parameter of the existing `VDGetDataRate` function.

Utility Functions

VDGetTimeCode

The `VDGetTimeCode` function instructs your video digitizer component to return timecode information for the incoming video signal.

```
pascal VideoDigitizerError VDGetTimeCode
                            (VideoDigitizerComponent ci,
                            TimeRecord *atTime,
                            const TimeCodeDef
                            *timeCodeFormat,
                            const TimecodeTime
                            *timeCodeTime);
```

`ci`	Specifies the video digitizer component for the request. Applications contains this reference from the Component Manager's `OpenComponent` function.
`atTime`	Specifies a location to receive the QuickTime movie time value corresponding to the timecode information.
`timeCodeFormat`	Contains a pointer to a timecode definition structure. Your video digitizer component returns the movie's timecode definition information.
`timeCodeTime`	Contains a pointer to a timecode record structure. Your video digitizer component returns the time value corresponding to the movie time contained in the `atTime` parameter.

DESCRIPTION

Typically, applications call this function once, at the beginning of a capture session.

For more information about the timecode data structures, see the "Movie Toollbox" chapter elsewhere in this document.

CHAPTER 7 MOVIE DATA EXCHANGE COMPONENTS

This chapter discusses new features in movie data exchange components.

DIRECT IMPORTATION

Some movie data import components can create a movie from a file without having to write to a separate disk file. Examples include MPEG and AIFF import components—data in files of these types can be played directly by the appropriate media handler components, without any data conversion. In such cases it is inappropriate for the user to have to specify a destination file, given that there is no need for such a file.

If you import component can operate in this manner, set the `canMovieImportInPlace` flag to 1 in your component flags when you register your component. The standard file dialog uses this flag to determine how to import files. The `OpenMovieFile` and `NewMovieFromFile` functions use this flag to open some kinds of files as movies.

AUDIO CD IMPORT COMPONENT

The Audio CD import component now creates AIFF files, rather than movie files. These files also contain movie resources, so you can open them as movies.

MOVIE DATA EXCHANGE COMPONENTS REFERENCE

Importing Movie Data

MovieImportGetFileType

The `MovieImportGetFileType` allows your movie data import component to tell the Movie Toolbox the appropriate file type for the most-recently imported movie file.

```
pascal ComponentResult MovieImportGetFileType
                                (MovieImportComponent ci,
                                OSType *fileType);
```

`ci`	Identifies the Movie Toolbox's connection to your movie data import component.
`fileType`	Contains a pointer to an `OSType` field. Your component should place the file type value that best identifies the movie data just imported. For example, Apple's Audio CD movie data import component sets this field to `'AIFF'` whenever it creates an AIFF file instead of a movie file.

DESCRIPTION

You should implement this function only if your movie data import component creates files other than QuickTime movies. By default, the Movie Toolbox makes new files movies, unless you override that default by providing this function.

ERROR CODES

`badComponentSelector`	0x80008002	Function not supported

CHAPTER 8 DERIVED MEDIA HANDLER COMPONENTS

This chapter discusses new features in derived media handler components.

DERIVED MEDIA HANDLER COMPONENTS REFERENCE

Functions

Managing Your Media Handler Component

MediaIdle

There is a minor change to the `MediaIdle` function that is related to the new media handler support for partial screen redrawing (for more information on this feature see the discussion of the `MediaGetDrawingRgn` function elsewhere in this chapter).

From time to time, your derived media handler component may determine that only a portion of the available drawing area needs to be redrawn. You can signal that condition to the base media handler component by setting the `mPartialDraw` flag to 1 in the flags your component returns to the Movie Toolbox from your `MediaIdle` function. You return these flags using the `flagsOut` parameter.

Whenever you set this flag to 1, the Movie Toolbox calls your component's `MediaGetDrawingRgn` function in order to determine the portion of the image that needs to be redrawn.

As an example, consider a full-screen animation. Only rarely is the entire image in motion. Typically, only a small portion of the screen image moves. By using partial redrawing, you can significantly improve the playback performance of such a movie.

Graphics Data Management

MediaGetDrawingRgn

The `MediaGetDrawingRgn` function allows your derived media handler component to specify a portion of the screen that must be redrawn. This region is defined in the movie's display coordinate system.

```
pascal ComponentResult MediaGetDrawingRgn (ComponentInstance
                                ci, RgnHandle *partialRgn);
```

`ci`	Identifies the Movie Toolbox's connection to your derived media handler.
`partialRgn`	Points to a handle that defines the screen region to be redrawn. Note that your component is responsible for disposing of this region once drawing is complete. Since the base media handler will use this region during redrawing, it is best to dispose of it when your component is closed.

DESCRIPTION

The Movie Toolbox calls this function in order to determine what part of the screen needs to be redrawn. By default, the Movie Toolbox redraws the entire region that belongs to your component. If your component determines that only a portion of the screen has changed, and has indicated this to the Movie Toolbox by setting the `mPartialDraw` flag to 1 in the `flagsOut` parameter of the `MediaIdle` function, the Movie Toolbox calls your component's `MediaGetDrawingRgn` function. Your component returns a region that defines the changed portion of the screen.

ERROR CODES

`badComponentSelector`	0x80008002	Function not supported
Memory Manager errors		

Base Media Handler Utility Functions

MediaForceUpdate

The MediaForceUpdate function allows your derived media handler component to influence when the base media handler updates the screen.

```
pascal ComponentResult MediaForceUpdate (ComponentInstance
                                ci, long forceUpdateFlags);
```

`ci` Identifies your derived media handler's connection to the base media handler.

`forceUpdateFlags` Specifies what you want the base media handler to do. The following flags are defined (be sure to set unused flags to 0):

`forceUpdateRedraw` Instructs the base media handler to call your derived media handler's `MediaIdle` function during the next `MoviesTask` execution. This allows your media handler to update the screen based on non-time-related events (typically you would get control only at sample changes). For example, you might want to highlight some text (say, a sample number) whenever the user stops the movie, even though this may not correspond to a sample change.

`forceUpdateNewBuffer` Instructs the base media handler to allocate a new off-screen buffer. This can be useful if you need to change the buffer's characteristics. The base media handler reallocates the buffer the next time the `MoviesTask` function is called.

CHAPTER 9 DATA HANDLER COMPONENTS

This chapter discusses data handler components. **Data handler components** allow the rest of QuickTime to retrieve time-based data from external storage devices and, in some cases, store time-based data on those devices.

This chapter is divided into the following sections:

- "About Data Handler Components" provides a general introduction to components of this type
- "Using Data Handler Components" discusses how QuickTime uses these components
- "Creating a Data Handler Component" describes how to create one of these components
- "Reference to Data Handler Components" presents detailed information about the functions that are supported by these components
- "Summary of Data Handler Components" contains a condensed listing of the constants, data structures, and functions supported by these components

This chapter addresses developers of data handler components, though it contains information for both developers and users of these components. If you plan to create a data handler component, you should read the entire chapter. If you are writing an application that uses components of this type, you should read the first two sections ("About Data Handler Components" and "Using Data Handler Components"), and then use the reference section as appropriate.

Furthermore, note that data handler components exist both in QuickTime for the Macintosh and QuickTime for Windows. Given that the architectures of these two systems are very similar, much of the background information is common to both environments—you will typically find this background information in *Inside Macintosh.* However, while the basic functionality and structure of these components is quite similar in both environments, there are some important technical differences. For example, the techniques you would use to create a component for Windows are quite different from those you would use on the Macintosh. Therefore, whenever appropriate, this chapter refers you to the specific *Inside Macintosh* or QuickTime for Windows documentation for additional information that is particular to these two environments.

> ***Note:*** This chapter describes the interface provided by data handler components. Note that this interface is supported only in QuickTime and QuickTime for Windows versions 2.0 or newer. In addition, unless noted otherwise, the data handler components supplied by Apple support the entire interface described in this note.

As components, data handler components rely on the facilities of the Component Manager. In order to create or use any component, your application must also use the Component Manager. If you are not familiar with the Component Manager, see "Component Manager" in *Inside Macintosh: More Macintosh Toolbox*. If you are developing for QuickTime for Windows, you should also take a look at *QuickTime for Windows: Components and Decompressors*. In addition, you should be familiar with the Movie Toolbox. See "Movie Toolbox" in *Inside Macintosh: QuickTime* for more information.

Note: Throughout this chapter, the terms *data handler* and *handler* refer to data handler components.

ABOUT DATA HANDLER COMPONENTS

This section provides background information about data handler components. After reading this section you should understand why these components exist and whether you need to create or use a data handler component.

Data Handler Components

Data handler components store and retrieve time-based data on behalf of other QuickTime components, typically media handler components. Figure 1 shows the logical relationships between applications, the Movie Toolbox, other QuickTime components, and data handlers during movie playback.

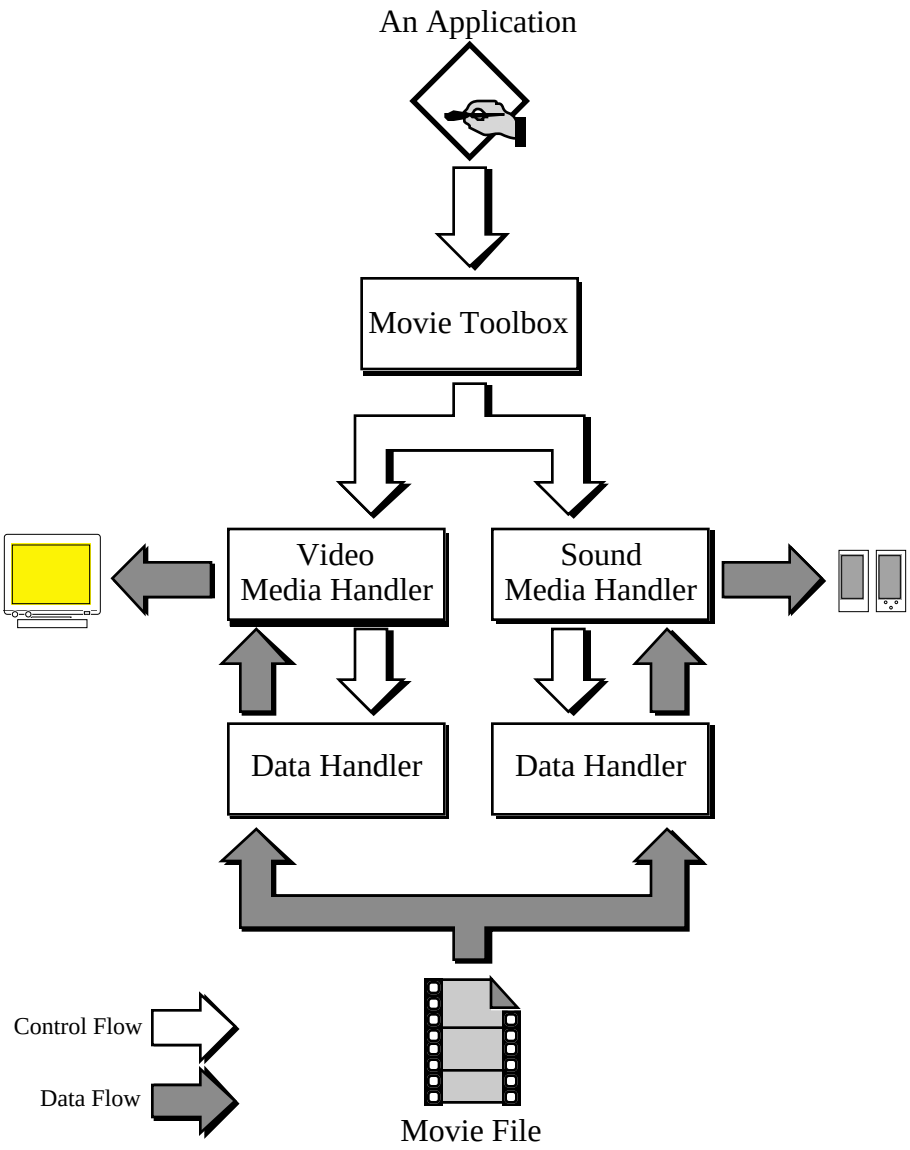


Figure 1: Playing a movie

Figure 2 shows the components that get involved in capturing movie data.

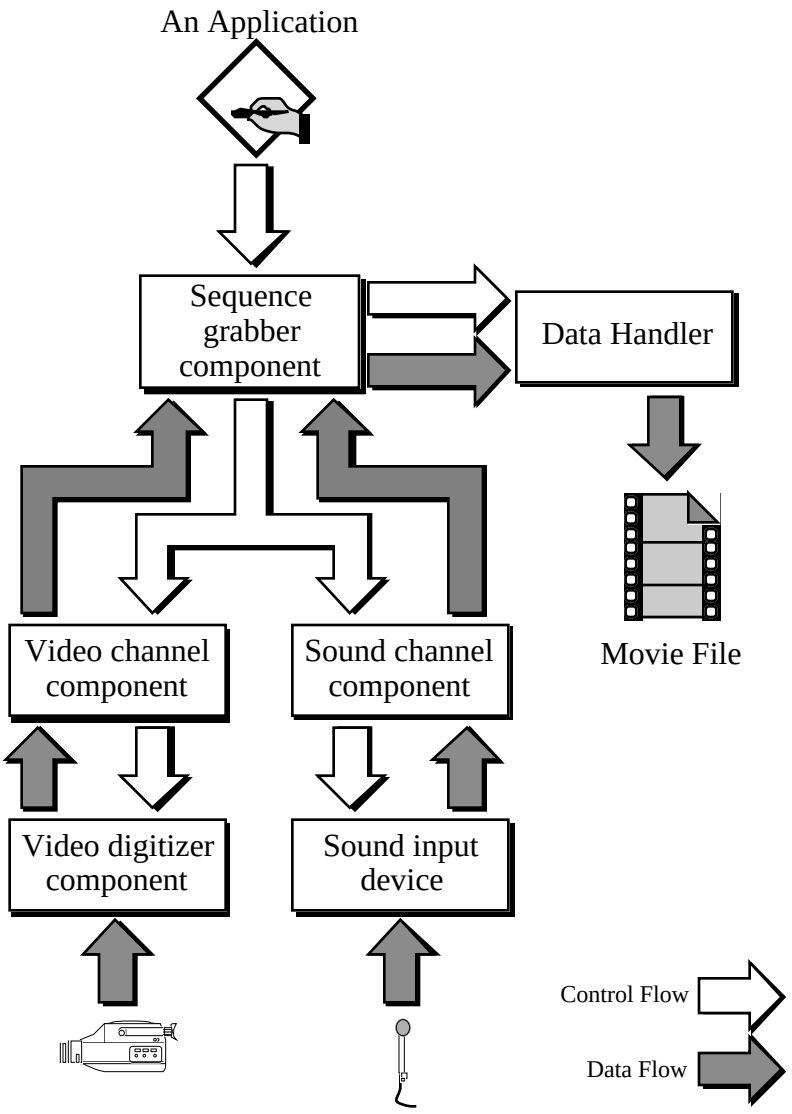


Figure 2: Capturing movie data

Data handlers isolate the rest of QuickTime from the details of how to store or retrieve time-based data from a particular storage medium. The data handler's primary function is to store or retrieve data at the time requested by the client program.

Data handlers do not know anything about the content of the data they process. It is the responsibility of the client (for example, a QuickTime media handler component) to process the data. In the case of a movie's video data, for example, the QuickTime media handler takes the data from a data handler and uses the facilities of the Image Compression Manager to display the movie data on the computer screen. See *Inside Macintosh: QuickTime Components* for more information about QuickTime media handler components.

While data handlers do not work with the content of the data they process, these components must be aware of all of the details involved in storing and retrieving data from the storage medium that they support. For example, Apple provides several data handlers. One supports data access from HFS volumes. Another, the memory-based data handler, allows QuickTime to retrieve movies from memory handles. These two data handler components use very different mechanisms to store and retrieve movie data. As a further example, in order to play movies from a multimedia server, you would need to use a data handler that understands the network protocols and data formats necessary to communicate with that server.

As is the case throughout QuickTime, all data handlers identify their movie-data containers with data references. The term **container** refers to the system element that contains the movie data and can be any element that can contain data. For example, a container may be an in-memory data structure, a local disk file, or a file on a networked multimedia server. Data references identify the location of the container and its type.

Different container types may require different types of references. For example, files are identified using aliases, while memory-based movies are identified by handles. The data reference data type is flexible enough to accommodate all these cases. It is up to each data handler component to specify the type of reference it requires, and to verify that the references supplied by client applications are valid. Data handler components use the component subtype value to specify the reference type they support.

Because the methods for accessing data on different devices may differ substantially, QuickTime supports multiple data handlers and a selection mechanism for choosing an appropriate handler. Whenever an application opens a movie container, the Movie Toolbox determines the most appropriate data handler component to use in order to access that container. The Movie Toolbox makes this determination by querying the various data handlers installed on the user's computer. If your application uses the Movie Toolbox, this selection process is transparent to your program. If you are developing a data handler, your component must support the selection functions (see "Reference to Data Handler Components," later in this chapter, for more information).

USING DATA HANDLER COMPONENTS

This section discusses how applications use data handler components. You should read this section if you are writing an application that uses these components or if you are creating your own data handler.

This section is divided into the following topics:

- "Selecting a Data Handler" discusses the facilities that are available to help your application choose the best data handler for a given context
- "Managing Data References" describes how your application goes about gaining access to a container using a data handler component
- "Retrieving Movie Data" talks about how your application reads movie data

- "Storing Movie Data" discusses how your application can write movie data using a data handler component
- "Managing the Data Handler" discusses your application's responsibilities while maintaining its connection with a data handler

Note that, if your application uses the Movie Toolbox to read and write movie data, you do not need to worry about the details of working with data handler components. The Movie Toolbox handles all of the data handler interactions for you. The information in this section is intended for developers whose applications need to work directly with data handler components.

Selecting a Data Handler

Before you can use a data handler component, your application must open a connection to that component. The easiest way to open a connection to a data handler component is to call the Move Toolbox's `GetDataHandler` function. You supply a data reference, and the Movie Toolbox selects an appropriate data handler component for you. For more information about this function, see the chapter "Movie Toolbox" in *Inside Macintosh: QuickTime*.

Alternatively, you may use the Component Manager to open your connection. Call the Component Manager's `OpenDefaultComponent` or `OpenComponent` function to do so.

In order to help developers choose the best data handler for a specific situation while still making it easy for an application to find a usable data handler, Apple has defined two separate and complementary mechanisms for selecting data handler components. The goal of these selection mechanisms is to ensure that your application is working with a data handler component that can process data from the movie container in question. Both mechanisms rely on characteristics of the current data reference in order to make the selection.

Selecting by Component Type Value

At the most basic level, your application can use the Component Manager's built-in selection mechanisms to find a data handler component for a data reference. You may use the Component Manager's `FindNextComponent` function in order to retrieve a list of all data handler components that meet your needs. You specify your request by supplying the component's characteristics in a component description record—in particular, in the `componentType`, `componentSubtype`, `componentManufacturer`, and `componentFlags` fields.

All data handler components have a component type value of `'dhlr'`, which is defined by the `dataHandlerType` constant. Data handler components use the value of the component subtype field to indicate the type of data reference they support. As a result of this convention, note that all data handlers that share a component subtype value must be able to recognize and work with data references of the same type. For example, file system data handlers always carry a component subtype value of `'alis'`, which indicates that their data references are file system aliases (note that this is true for QuickTime on the Macintosh and under Windows, even though there is not, properly, a file system alias under Windows). Apple's memory-based data handler for the Macintosh has a component subtype value of `'hndl'`.

Apple has not defined any special manufacturer field values or component flags values for data handler components. You may use the manufacturer field to select data handlers supplied by a specific vendor. To do so, you would need to determine the appropriate manufacturer field value for that vendor.

Interrogating a Data Handler's Capabilities

While you can use the Component Manager's selection mechanisms to find a data handler component that can recognize data references of a specific type, your application must interact with the data handler in order to determine whether it can support a specific data reference. Apple has defined two functions that allow you to query a data handler component in order to find out whether it can work with a data reference. By using these two functions, your application can choose a data handler that is best-suited to its specific needs.

Before you can use either of these functions, your application must open a connection to the data handler component, using the Component Manager.

Using the `DataHCanUseDataRef` function, you supply a data reference to the data handler component. The component then reports what it can do with that data reference. The returned value indicates the level and, to some extent, the quality of service the data handler can provide (for example, whether the component can read data from or write data to the data reference, and whether the component uses any special support when working with that data reference).

Because calling the `DataHCanUseDataRef` function in several data handlers can get time consuming, Apple has also defined a function that helps narrow the search somewhat. Using the `DataHGetVolumeList` function, your application can obtain a list of all of the file system volumes that a data handler can support. In response to your request, the data handler returns a list of all of the volumes it can support, along with flags indicating the level and quality of service the data handler can provide for containers on that volume.

For more information on these functions, see "Selecting a Data Handler," later in this chapter.

Managing Data References

Once you have selected a data handler component, you must provide a data reference to the data handler. Use the `DataHSetDataRef` function to supply a data reference to a data handler. Once you have assigned a data reference to the data handler, your application may start reading and/or writing movie data from that data reference. The `DataHGetDataRef` function allows your application to obtain a data handler's current data reference.

Data handlers also provide a function that allows your application to determine whether two data references are equivalent (that is, refer to the same movie container). Your application provides a data reference to the `DataHCompareDataRef` function. The data handler returns a Boolean value indicating whether that data reference matches the data handler's current data reference.

For more information on these functions, see "Working With Data References," later in this chapter.

Retrieving Movie Data

Before your application can read data using a data handler component, you must open a read path to the current data reference. Use the `DataHOpenForRead` function to request read access to the current data reference. Once you have gained read access to the data reference, data handlers provide both high- and low-level read functions.

The high-level function, `DataHGetData`, provides an easy-to-use, synchronous read interface. Being a synchronous function, `DataHGetData` does not return control to your application until the data handler has read and delivered the data you request.

If you need more control over the read operation, you can use the low-level function, `DataHScheduleData`, to issue asynchronous read requests. When you call this function, you provide detailed information specifying when you need the data from the request. The data handler returns control to your application immediately, and then processes the request when appropriate. When the data handler completes the request, it calls your data-handler completion function to report that the request has been satisfied (see "Completion Function" for more information on the data-handler completion function).

Besides simply scheduling read requests that must be satisfied during a movie's playback, another use of the `DataHScheduleData` function is to prepare a movie for playback (commonly referred to as pre-rolling the movie). The `DataHScheduleData` function uses several special values to indicate a pre-roll operation. Your application calls the `DataHScheduleData` function one or more times to schedule the pre-roll read requests, and then uses the `DataHFinishData` function to tell the data handler to start delivering the requested data.

For more information on these functions and about pre-roll operations, see "Reading Movie Data," later in this chapter.

Storing Movie Data

Before your application can write data using a data handler component, you must open a write path to the current data reference. Use the `DataHOpenForWrite` function to request write access to the current data reference. Once you have gained write access to the data reference, data handler components provide both high- and low-level write functions.

Note: QuickTime for Windows does not support writing movie data.

The high-level function, `DataHPutData`, allows you to easily append data to the end of the container identified by a data reference. Except when capturing movie data using the sequence grabber component, the Movie Toolbox uses this call when writing data to movie files. However, this function does not allow your application to write to any location other than the end of the container. In addition, this is a synchronous operation, so control is not returned to your program until the write is complete. As a result, this function is not well-suited to high-performance write operations, such as would be required to capture a movie.

If you need a more flexible write facility, or one with higher performance characteristics, you can use the `DataHWrite` function. This function is intended to support high-speed writes, suitable for movie capture operations. For example, Apple's sequence grabber component uses this data handler function to capture movies.

When you call this function, you provide detailed information specifying the location in the container that is to receive the data. The data handler returns control to your application immediately, and then processes the request asynchronously. When the data handler completes the request, it calls your data-handler completion function to report that the request has been satisfied (see "Completion Function" for more information on the data-handler completion function).

In addition to the `DataHWrite` function, data handler components provide several other "helper" functions that allow you to create new movie containers and prepare them for a movie capture operation.

For more information on all of these functions, see "Writing Movie Data," later in this chapter.

Managing the Data Handler

Data handler components provide a number of functions that your application can use to manage its connection to the handler. The most important among these is `DataHTask`, which provides processor time to the handler. Your application should call this function often so that the handler has enough time to do its work.

Other functions in this category provide playback hints to the data handler and allow your application to influence how the component handles its cached data.

For more information on these functions, see "Managing Data Handler Components," later in this chapter.

CREATING A DATA HANDLER COMPONENT

This section discusses the details of creating a data handler component and includes source code for a simple data handler component. After reading this section, you will understand all of the special requirements of these components. The functional interface that your component must support is described in "Reference to Data Handler Components," later in this chapter.

You should consider developing your own data handler component if you are planning to provide a new type of movie container or a container that requires special data handling techniques. For example, if you are planning to develop a networked multimedia server, you would most likely need to develop a new data handler that could support the special protocols required by your server. By encapsulating that protocol support in a data handler, QuickTime applications can access the movie data on your server without having to implement any special support. In this way, your server becomes a seamless part of the user's system.

Before reading this section, you should be familiar with how to create components. See "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for a complete discussion of components, how to use them, and how to create them on the Macintosh. For further information about using the Component Manager with QuickTime for Windows, see *QuickTime for Windows: Components and Decompressors.*

General Information

All data handler components have a component type value of `'dhlr'`, which is defined by the `dataHandlerType` constant. Data handler components use the value of the component subtype field to indicate the type of data reference they support. As a result of this convention, note that all data handlers that share a component subtype value must be able to recognize and work with data references of the same type. For example, file system data handlers always carry a component subtype value of `'alis'`, which indicates that their data references are file system aliases (note that this is true for QuickTime on the Macintosh and under Windows, even though there is not, properly, a file system alias under Windows). Apple's memory-based data handler for the Macintosh has a component subtype value of `'hndl'`.

```
#define dataHandlerType 'dhlr'
#define rAliasType      'alis'
```

Apple has not defined any special manufacturer field values or component flags values for data handler components. Developers may use the manufacturer field value to select your data handler from among all the data handlers that support a given type of data reference.

Apple has defined a functional interface for data handler components. For information about the functions that your component must support, see "Reference to Data Handler Components" later in this chapter. You can use the following constants to refer to the request codes for each of the functions that your component must support:

```
enum {

   kDataGetDataSelector                = 2,     /* DataHGetData */
   kDataPutDataSelector                = 3,     /* DataHPutData */
   kDataFlushDataSelector              = 4,     /* DataHFlushData */
   kDataOpenForWriteSelector           = 5,     /* DataHOpenForWrite */
   kDataCloseForWriteSelector          = 6,     /* DataHCloseForWrite */
   kDataOpenForReadSelector            = 8,     /* DataHOpenForRead */
   kDataCloseForReadSelector           = 9,     /* DataHCloseForRead */
   kDataSetDatRefSelector              = 10,    /* DataHSetDataRef */
   kDataGetDataRefSelector             = 11,    /* DataHGetDataRef */
   kDataCompareDataRefSelector         = 12,    /* DataHCompareDataRef
*/
   kDataTaskSelector                   = 13,    /* DataHTask */
   kDataScheduleDataSelector           = 14,    /* DataHScheduleData */
   kDataFinishDataSelector             = 15,    /* DataHFinishData */
   kDataFlushCacheSelector             = 16,    /* DataHFlushCache */
   kDataResolveDataRefSelector         = 17,    /* DataHResolveDataRef
*/
   kDataGetFileSizeSelector            = 18,    /* DataHGetFileSize */
   kDataCanUseDataRefSelector          = 19,    /* DataHCanUseDataRef */
   kDataGetVoumeListSelector           = 20,    /* DataHGetVolumeList */
   kDataWriteSelector                  = 21,    /* DataHWrite */
   kDataPreextendSelector              = 22,    /* DataHPreextend */
   kDataSetFileSizeSelector            = 23,    /* DataHSetFileSize */
   kDataGetFreeSpaceSelector           = 24,    /* DataHGetFreeSpace */
   kDataCreateFileSelector             = 25,    /* DataHCreateFile */
   kDataGetPreferredBlockSizeSelector  = 26,    /*
DataHGetPreferredBlockSize */
   kDataGetDeviceIndexSelector         = 27,    /* DataHGetDeviceIndex
*/
   /* 28 and 29 unused */
   kDataGetScheduleAheadTimeSelector   = 30,    /*
                                          DataHGetScheduleAheadTime */
   kDataSetOSFileRefSelector           = 516,   /* DataHSetOSFileRef */
   kDataGetOSFileRefSelector           = 517,   /* DataHGetOSFileRef */

   kDataPlaybackHintsSelector       = 3+0x100   /* DataHPlaybackHints */
};
```

Macintosh Data Handler Components

This section provides sample code for a working Macintosh data handler component.

Sample Macintosh Data Handler

```
#include <Aliases.h>
#include <Files.h>
#include <OSUtils.h>
```

```
#include “DataHandlerPrototypes.h”

// these selectors belong in the header file

enum {DataGetDataSelector = 2 };
enum {DataPutDataSelector = 3 };
enum {DataOpenForWriteSelector = 5 };
enum {DataCloseForWriteSelector = 6 };
enum {DataOpenForReadSelector = 8 };
enum {DataCloseForReadSelector = 9 };
enum {DataSetAliasSelector = 10 };
enum {DataGetAliasSelector = 11 };
enum {DataCompareAliasSelector = 12 };
enum {DataTaskSelector = 13 };
enum {DataScheduleDataSelector = 14 };
enum {DataCanUseDataRef = 19 };
enum {DataGetVolumeListSelector = 20 };

// data structures

typedef struct {
   ComponentInstance    self;

   AliasHandle       alias;

   short          readFref;
   short          writeFref;
} DataHandlerGlobalsRecord, *DataHandlerGlobals;

// function declarations

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

ComponentFunctionUPP DHSelectorLookup(short selector);

pascal ComponentResult DHOpen(DataHandlerGlobals storage,
                            ComponentInstance self);
pascal ComponentResult DHClose(DataHandlerGlobals storage,
                            ComponentInstance self);
pascal ComponentResult DHCanDo(DataHandlerGlobals storage,
                            short functionSelector);
pascal ComponentResult DHVersion( DataHandlerGlobals storage);

pascal ComponentResult DHGetData(DataHandlerGlobals storage, Handle h,
                            long offsetIntoHandle, long offset,
                            long size);
pascal ComponentResult DHPutData(DataHandlerGlobals storage, Handle h,
                            long hOffset, long *offset, long size);

pascal ComponentResult DHSetAlias(DataHandlerGlobals storage,
                            AliasHandle alias);
pascal ComponentResult DHGetAlias(DataHandlerGlobals storage,
                            AliasHandle *alias);
```

```
pascal ComponentResult DHCompareAlias(DataHandlerGlobals storage,
                              AliasHandle alias, Boolean *equal);

pascal ComponentResult DHScheduleData (DataHandlerGlobals storage,
                              Ptr dataPtr,long fileOffset,
                              long dataSize, long refCon,
                              TimeRecord *timeNeededBy,
                              DataHCompletionUPP completionRoutine);

pascal ComponentResult DHOpenForRead(DataHandlerGlobals storage);
pascal ComponentResult DHCloseForRead(DataHandlerGlobals storage);
pascal ComponentResult DHOpenForWrite(DataHandlerGlobals storage);
pascal ComponentResult DHCloseForWrite(DataHandlerGlobals storage);

pascal ComponentResult DHGetVolumeList(DataHandlerGlobals storage,
                              DataHVolumeList *volumeList);
pascal ComponentResult DHCanUseDataRef(DataHandlerGlobals storage,
                              Handle dataRef, long *useFlags);

// main function

pascal ComponentResult main(ComponentParameters *params, Handle storage)
{
   ComponentResult err;
   ComponentFunctionUPP componentProc;

   componentProc = DHSelectorLookup(params->what);

   if (componentProc)
      err = CallComponentFunctionWithStorage(storage, params,
                                             componentProc);
   else
      err = badComponentSelector;

   return err;
}

// determine function based on selected request

ComponentFunctionUPP DHSelectorLookup(short selector)
{
   ComponentFunctionUPP componentProc = 0;

   switch (selector) {
      case kComponentVersionSelect:
         componentProc = (ComponentFunctionUPP)DHVersion;
         break;
      case kComponentCanDoSelect:
         componentProc = (ComponentFunctionUPP)DHCanDo;
         break;
      case kComponentCloseSelect:
         componentProc = (ComponentFunctionUPP)DHClose;
         break;
      case kComponentOpenSelect:
```

```
            componentProc = (ComponentFunctionUPP)DHOpen;
            break;

         case DataGetDataSelector:
            componentProc = (ComponentFunctionUPP)DHGetData;
            break;
         case DataPutDataSelector:
            componentProc = (ComponentFunctionUPP)DHPutData;
            break;
         case DataOpenForReadSelector:
            componentProc = (ComponentFunctionUPP)DHOpenForRead;
            break;
         case DataCloseForReadSelector:
            componentProc = (ComponentFunctionUPP)DHCloseForRead;
            break;
         case DataOpenForWriteSelector:
            componentProc = (ComponentFunctionUPP)DHOpenForWrite;
            break;
         case DataCloseForWriteSelector:
            componentProc = (ComponentFunctionUPP)DHCloseForWrite;
            break;
         case DataSetAliasSelector:
            componentProc = (ComponentFunctionUPP)DHSetAlias;
            break;
         case DataGetAliasSelector:
            componentProc = (ComponentFunctionUPP)DHGetAlias;
            break;
         case DataCompareAliasSelector:
            componentProc = (ComponentFunctionUPP)DHCompareAlias;
            break;
         case DataScheduleDataSelector:
            componentProc = (ComponentFunctionUPP)DHScheduleData;
            break;
         case DataCanUseDataRef:
            componentProc = (ComponentFunctionUPP)DHCanUseDataRef;
            break;
         case DataGetVolumeListSelector:
            componentProc = (ComponentFunctionUPP)DHGetVolumeList;
            break;
      }

      return componentProc;
}

// open data handler connection

pascal ComponentResult DHOpen(DataHandlerGlobals storage,
                              ComponentInstance self)
{
      ComponentResult err;

      storage =
(DataHandlerGlobals)NewPtrClear(sizeof(DataHandlerGlobalsRecord));
      if (err = MemError())
```

```
      return err;

   storage->self = (ComponentInstance)self;

   SetComponentInstanceStorage(storage->self,(Handle)storage);

   return noErr;
}

// close component connection

pascal ComponentResult DHClose(DataHandlerGlobals storage,
                               ComponentInstance self)
{
   if (storage != nil) {
      DHCloseForRead(storage);
      DHCloseForWrite(storage);

      if (storage->alias != nil)
         DisposeHandle((Handle)storage->alias);

      DisposePtr((Ptr)storage);
   }

   return noErr;
}

// determine whether data handler supports request

pascal ComponentResult DHCanDo(DataHandlerGlobals storage,
                               short functionSelector)
{
   return DHSelectorLookup(functionSelector) != 0;
}

// return component's version

pascal ComponentResult DHVersion(DataHandlerGlobals storage)
{
   return 0x00020001;
}

// read data

pascal ComponentResult DHGetData(DataHandlerGlobals storage, Handle h,
                               long offsetIntoHandle, long offset,
long size)
{
   OSErr          err;
   SignedByte     saveState;

   if (!storage->readFref) {
      err = DHOpenForRead(storage);
      if (err != noErr)
```

```
         return err;
   }

   saveState = HGetState(h);
   HLock(h);
   err = DHScheduleData(storage, *h + offsetIntoHandle,
                           offset, size, 0, nil, nil);
   HSetState(h, saveState);

   return err;
}

// write data

pascal ComponentResult DHPutData(DataHandlerGlobals storage, Handle h,
                                 long hOffset, long *offset, long size)
{
   OSErr          err;

   if (!storage->writeFref) {
      err = DHOpenForWrite(storage);
      if (err != noErr)
         return err;
   }

   err = SetFPos(storage->writeFref, fsFromLEOF, 0);
   if (err == noErr) {
      if (offset)
         err = GetFPos(storage->writeFref, offset);
      if (err == noErr)
         err = FSWrite(storage->writeFref, &size, *h + hOffset);
   }

   return err;
}

// set alias

pascal ComponentResult DHSetAlias(DataHandlerGlobals storage,
                                  AliasHandle alias)
{
   OSErr err = noErr;

   // throw away the old one
   if (storage->alias) {
      DisposeHandle((Handle)storage->alias);
      storage->alias = nil;
   }

   // copy the new one, if there is one
   if (alias) {
      err = HandToHand((Handle *)&alias);
      if (err == noErr)
         storage->alias = alias;
```

```
   }

   return err;
}

// retrieve alias

pascal ComponentResult DHGetAlias(DataHandlerGlobals storage,
                                  AliasHandle *alias)
{
   OSErr err = noErr;

   *alias = nil;
   if (storage->alias) {
      *alias = storage->alias;
      err = HandToHand((Handle *)alias);
   }

   return err;
}

// compare two aliases

pascal ComponentResult DHCompareAlias(DataHandlerGlobals storage,
                                      AliasHandle alias, Boolean
*equal)
{
   OSErr err = paramErr;
   FSSpec fss1, fss2;
   Boolean whoCares;

   *equal = false;

   if (storage->alias && alias) {
      err = ResolveAlias(nil, storage->alias, &fss1, &whoCares);
      if (err == noErr) {
         err = ResolveAlias(nil, alias, &fss2, &whoCares);
         if (err == noErr) {
            *equal = (fss1.vRefNum == fss2.vRefNum) &&
                     (fss1.parID == fss2.parID) &&
                     EqualString(fss1.name, fss2.name, false, false);
         }
      }
   }

   return err;
}

// scheduled read

pascal ComponentResult DHScheduleData(DataHandlerGlobals storage,
                                      Ptr dataPtr,long fileOffset,
                                      long dataSize, long refCon,
                                      TimeRecord *timeNeededBy,
```

```
                                        DataHCompletionUPP
completionRoutine)
{
   OSErr err;

   if (storage->readFref == 0) {
      err = DHOpenForRead(storage);
      if (err)
         return err;
   }

   err = SetFPos(storage->readFref, fsFromStart, fileOffset);
   if (err == noErr)
      err = FSRead(storage->readFref, &dataSize, dataPtr);

   // Always call completion routine, even on an error.
   if (completionRoutine != nil)
      (*completionRoutine)(dataPtr, refCon, err);

   return err;
}

// open container for read

pascal ComponentResult DHOpenForRead(DataHandlerGlobals storage)
{
   OSErr err;
   FSSpec fss;
   Boolean whoCares;

   if (storage->readFref != 0)
      return noErr;

   if (storage->alias == nil)
      return dataNoDataRef;

   err = ResolveAlias(nil, storage->alias, &fss, &whoCares);
   if (err) return err;

   err = FSpOpenDF(&fss, fsRdPerm, &storage->readFref);

   return err;
}

// close container after reading

pascal ComponentResult DHCloseForRead(DataHandlerGlobals storage)
{
   if (storage->readFref) {
      FSClose(storage->readFref);
      storage->readFref = 0;
   }

   return noErr;
```

```
}

// open container for write

pascal ComponentResult DHOpenForWrite(DataHandlerGlobals storage)
{
   OSErr err;
   FSSpec fss;
   Boolean whoCares;

   if (storage->writeFref != 0)
      return noErr;

   if (storage->alias == nil)
      return dataNoDataRef;

   err = ResolveAlias(nil, storage->alias, &fss, &whoCares);
   if (err) return err;

   err = FSpOpenDF(&fss, fsRdWrPerm, &storage->writeFref);

   return err;
}

// close container after writing

pascal ComponentResult DHCloseForWrite(DataHandlerGlobals storage)
{
   if (storage->writeFref) {
      FSClose(storage->writeFref);
      storage->writeFref = 0;
   }

   return noErr;
}

//
// This function limits the set of drives this data handler will be used
to
// read from to those with names beginning with the letter Q.
//
Boolean isVRefNumOK(short vRefNum);
Boolean isVRefNumOK(short vRefNum)
{
   ParamBlockRec pb;
   Str63 name;

   name[0] = 0;
   pb.volumeParam.ioVolIndex = 0;
   pb.volumeParam.ioVRefNum = vRefNum;
   pb.volumeParam.ioNamePtr = name;
   if (PBGetVInfoSync(&pb) != noErr)
      return false;
```

```
   return (name[1] == 'Q') || (name[1] == 'q');
}

// determine whether we can handle the data reference

pascal ComponentResult DHCanUseDataRef(DataHandlerGlobals storage,
                                       Handle dataRef, long *useFlags)
{
   OSErr err;
   FSSpec fss;
   Boolean whoCares;

   *useFlags = 0;

   err = ResolveAlias(nil, (AliasHandle)dataRef, &fss, &whoCares);
   if (err) return err;

   if (isVRefNumOK(fss.vRefNum))
      *useFlags = kDataHCanRead | kDataHSpecialRead | kDataHCanWrite;

   return noErr;
}

//
// This call is only required for data handlers with a subtype of
// rAliasType ('alis').
//
pascal ComponentResult DHGetVolumeList(DataHandlerGlobals storage,
                                       DataHVolumeList *volumeList)
{
   OSErr err = noErr;
   DataHVolumeList list;
   VCB *vq;

   list = (DataHVolumeList)NewHandle(0);
   if (err = MemError())
      goto bail;

   vq = (VCB *)GetVCBQHdr()->qHead;
   while (vq) {
      if (isVRefNumOK(vq->vcbVRefNum)) {
         DataHVolumeListRecord vlr;

         // add it to our list
         vlr.vRefNum = vq->vcbVRefNum;
         vlr.flags = kDataHCanRead | kDataHSpecialRead | kDataHCanWrite;
         err = PtrAndHand((Ptr)&vlr, (Handle)list, sizeof(vlr));
         if (err)
            goto bail;
      }
      vq = (VCB *)vq->qLink;
   }

bail:
```

```
   if (err) {
      DisposeHandle((Handle)list);
      list = nil;
   }
   *volumeList = list;
   return err;
}
```

Windows Data Handler Components

This section discusses additional information you need to know before you develop your own Windows data handler component. It also includes source code for a Windows data handler component.

While data handler components to be used with QuickTime for Windows are functionally quite similar to Macintosh data handlers, there are some differences you need to consider before developing your own Windows data handler. First of all, QuickTime for Windows does not support a write data path. Therefore, your data handler needs to support only those functions that allow QuickTime to read movie data.

In addition, Windows components are build as special dynamic link libraries (DLLs). You need to structure your code appropriately.

Sample Windows Data Handler

```
/*
******************************************************************************
**
** File: datah.cpp
**
** Description:
**
** Data Handler component for QuickTime for Windows.
**
** Routines:
**
** Routines enclosed in [brackets] exist, but are unsupported.
**
** DataHOpen();                  - component manager open call
** DataHClose();                 - component manager close call
** DataHCanDo();                 - component manager cando call
** DataHVersion();               - component manager version call
** DataHGetData();               - immediate data read
** [DataHPutData();]             - data write
** [DataHFlushData();]           - flush write buffers
** [DataHOpenForWrite();]        - open for write access
** [DataHCloseForWrite();]       - close for write access
** DataHOpenForRead();           - open for read access
** DataHCloseForRead();          - close for read access
** DataHSetDatRef();             - set data reference
```

```
** DataHGetDataRef();            - get data reference
** DataHCompareDataRef();        - compare data references
** DataHTask();                  - provide background time
** DataHScheduleData();          - schedule advance read
** DataHFinishData();            - complete scheduled reads
** DataHFlushCache();            - flush cache buffers
** DataHResolveDataRef();        - resolve data reference
** DataHGetFileSize();           - return file size
** DataHCanUseDataRef();         - check if data ref can be used
** DataHGetVolumeList();         - return list of volumes supported
** [DataHWrite();]               - write data
** [DataHPreextend();]           - extend file
** [DataHSetFileSize();]         - set file size
** DataHGetFreeSpace();          - get device free space
** [DataHCreateFile();]          - create file
** DataHGetPreferredBlockSize(); - get preferred block size
** DataHGetDeviceIndex();        - get unique device index
** DataHGetScheduleAheadTime();  - get preferred advance read time
** DataHPlaybackHints();         - provide data ref playback hints
** DataHSetOSFileReference();    - set HFILE as data reference
** DataHGetOSFileReference();    - get references from SetOSFile...
** _DataHDirectRead();           - direct device read function
**
*************************************************************************/
// Windows header files
#include <windows.h>
#include <windowsx.h>
#include <mmsystem.h>

// dos headers
#include <direct.h>
#include <dos.h>

// Compiler header files
#include <string.h>
#include <ctype.h>
#include <stdlib.h>

// Application header files
#define INTERNAL_DHLR
#include “datahp.h”
#include <qtdebug.h>

// Apple Computer four character type.
#define ostypeAPPL QTFOURCC('a','p','p','l')

// macros
#define DATAHPARM(x,y) GetPrivateProfileInt(“Data
Handler”,x,y,QTW_PROFILE)
#define DEBUG TRUE

// prototypes for external functions
DWORD QTAPI DataHEntry(void);
```

```
// Global data
ComponentDescription cdTable =              // one component in this DLL
  { ostypeDHLR                              // ostypeComponentType
  , ostypeHNDL                              // ostypeComponentSubType
  , ostypeAPPL                              //
ostypeComponentManufacturer
  , 0                                       // dwComponentFlags
  , 0                                       // dwComponentFlagsMask
  , ( ComponentRoutine) DataHEntry          // crEntryPoint
  , 0                                       // hrsrcName
  , 0                                       // hrsrcInfo
  , 0                                       // hrsrcIcon
  } ;


/*
**************************************************************************
**
** Name: DataHOpen()
**
** Description:
**
** Opens an instance of the component.
**
** The general data handler initialization is done here, so that any
memory
** used will not be allocated until an instance of the data handler is
actually
** opened.
**
**************************************************************************
*/
ComponentResult QTAPI DataHOpen( STKOFF_CMP so, ComponentInstance ci)
{
   void far *storageH, far *globalH;
   DataHInstanceStoragePtr storage;
   DataHGlobalStoragePtr globals;

   // allocate the cross-instance globals
   globalH = (void far *)GetComponentRefcon(ci);
   if(globalH == NULL)
   {
      // allocate global storage
      globalH = (void far *)GlobalAlloc(GMEM_ZEROINIT,
                                        sizeof(DataHGlobalStorage));
      if(globalH == NULL)
         return insufficientMemory;

      // set the refcon so that we know we have been initialized
      SetComponentRefcon(ci, (long)globalH);
   }

   globals = (DataHGlobalStoragePtr)GlobalLock((const void
                                              near *)LOWORD(globalH));
```

```
   if(globals == NULL)
   {
      GlobalFree((const void near *)LOWORD(globalH));
      return(insufficientMemory);
   }


   // allocate instance storage
   storageH = (void far *)GlobalAlloc(GMEM_ZEROINIT,
                                      sizeof(DataHInstanceStorage));
   if(storageH == NULL)
      return insufficientMemory;
   storage = (DataHInstanceStoragePtr)GlobalLock((const void
                                                  near
*)LOWORD(storageH));
   if(storage == NULL)
   {
      GlobalUnlock((const void near *)LOWORD(globalH));
      GlobalFree((const void near *)LOWORD(storageH));
      return(insufficientMemory);
   }

   // init storage fields
   storage->ci = ci;
   wwList_Init(&storage->readRequestList);
   wwList_InitCache(&storage->readRequestList, 10);   // init node cache

   // set storage for this component
   SetComponentInstanceStorage(ci, (LPVOID)storageH);

   // done
   GlobalUnlock((const void near *)LOWORD(storageH));
   GlobalUnlock((const void near *)LOWORD(globalH));
   return(noErr);
}


/*
**************************************************************************
**
** Name: DataHClose()
**
** Description:
**
** Closes an instance of the component.
**
**************************************************************************
*/
ComponentResult QTAPI DataHClose( STKOFF_CMP so, ComponentInstance ci)
{
   void far *storageH, far *globalH;
   DataHInstanceStoragePtr storage;

   // locate instance storage
```

```
   storageH = GetComponentInstanceStorage(ci);
   if(storageH != NULL)
   {
      storage = (DataHInstanceStoragePtr)GlobalLock((const void
                                                     near
*)LOWORD(storageH));
      if(storage != NULL)
      {
         // close the file
         if (storage->fileRefNum)
            mmioClose(storage->fileRefNum, 0);

         // release memory allocated for filename
         if(storage->fileName)
            GlobalFree((HGLOBAL)storage->fileName);

         // release memory for instance storage
         GlobalUnlock((const void near *)LOWORD(storageH));
         GlobalFree((const void near *)LOWORD(storageH));
      }
   }

   // release global storage if this is the last instance of the
component
   if(CountComponentInstances(ci) == 1)
   {
      globalH = (void far *)GetComponentRefcon(ci);
      if(globalH)
      {
         GlobalFree((const void near *)LOWORD(globalH));
         SetComponentRefcon(ci,NULL);
      }
   }

   return(noErr);
}


/*
**********************************************************************
**
** Name: DataHCanDo()
**
** Description:
**
** Returns TRUE if a call is supported.
**
**********************************************************************
*/
ComponentResult QTAPI DataHCanDo( STKOFF_CMP so, long lFunctionSelector)
{
   switch (lFunctionSelector)
   {
      /* standard component manager calls */
```

```
      case kDataVersionSelector:
      case kDataCanDoSelector:
      case kDataCloseSelector:
      case kDataOpenSelector:

      /* data handler calls */
      case kDataGetDataSelector:
      case kDataOpenForReadSelector:
      case kDataCloseForReadSelector:
      case kDataSetDatRefSelector:
      case kDataGetDataRefSelector:
      case kDataCompareDataRefSelector:
      case kDataTaskSelector:
      case kDataScheduleDataSelector:
      case kDataFinishDataSelector:
      case kDataFlushCacheSelector:
      case kDataResolveDataRefSelector:
      case kDataGetFileSizeSelector:
      case kDataCanUseDataRefSelector:
      case kDataGetVolumeListSelector:
      case kDataPlaybackHintsSelector:
      case kDataSetOSFileReferenceSelector:
      case kDataGetOSFileReferenceSelector:
         return(TRUE);
         break;
      default:
         return(FALSE);
         break;
   }

   return(FALSE);
}


/*
**************************************************************************
**
** Name: DataHVersion()
**
** Description:
**
** Returns version number of the component.
**
**************************************************************************
*/
ComponentResult QTAPI DataHVersion( STKOFF_CMP so, ComponentInstance ci)
{
   return(kDataHVersion);
}


/*
**************************************************************************
**
```

```
** Name: DataHGetData()
**
** Description:
**
** Synchronous data read.
**
*****************************************************************************
*/
ComponentResult QTAPI DataHGetData (DHLR_FPARM1
   Handle h,      // handle to destination of data
   long hOffset,  // offset into handle to place data
   long offset,   // offset within file of data to read
   long size)     // amount of data to read
{
   void far *storageH = instanceStorage;
   DataHInstanceStoragePtr storage;
   char *dataPtr;

   // lock the storage
   storage = (DataHInstanceStoragePtr)GlobalLock((const void
                                                  near
*)LOWORD(storageH));
   if(storage == NULL)
      return(insufficientMemory);

   // verify that we have an open file
   if(storage->fileRefNum == 0)
   {
      GlobalUnlock((const void near *)LOWORD(storageH));
      return(dataNotOpenForRead);
   }

   // lock and deref the user handle
   dataPtr = (char *)GlobalLock(h);
   if(dataPtr == NULL)
   {
      GlobalUnlock((const void near *)LOWORD(storageH));
      return(insufficientMemory);
   }

   // build the pointer to the destination area
   dataPtr += hOffset;

   if(_DataHDirectRead(storage, dataPtr, offset, size) == FALSE)
   {
      GlobalUnlock(h);
      GlobalUnlock((const void near *)LOWORD(storageH));
      return(dataReadErr);
   }

   // done
   GlobalUnlock(h);
   GlobalUnlock((const void near *)LOWORD(storageH));
   return(noErr);
```

```
}



/*
**************************************************************************
**
** Name: DataHPutData()
**
** Description:
**
** Synchronous write.  Not supported.
**
**************************************************************************
*/
ComponentResult QTAPI DataHPutData (DHLR_FPARM1
   Handle h,
   long hOffset,
   long *offset,
   long size)
{
   return(badComponentSelector);
}


/*
**************************************************************************
**
** Name: DataHFlushData()
**
** Description:
**
** Flush unwritten data.  Not supported.
**
**************************************************************************
*/
ComponentResult QTAPI DataHFlushData (DHLR_FPARM2)
{
   return(badComponentSelector);
}


/*
**************************************************************************
**
** Name: DataHOpenForWrite()
**
** Description:
**
** Open data reference for write access.  Not supported.
**
**************************************************************************
```

```
*/
ComponentResult QTAPI DataHOpenForWrite (DHLR_FPARM2)
{
   return(badComponentSelector);
}

/*
**************************************************************************
**
** Name: DataHCloseForWrite()
**
** Description:
**
** Close data reference that has been opened for write access.  Not
supported.
**
**************************************************************************
*/
ComponentResult QTAPI DataHCloseForWrite (DHLR_FPARM2)
{
   return(badComponentSelector);
}


/*
**************************************************************************
**
** Name: DataHOpenForRead()
**
** Description:
**
** Open data reference for read access.
**
**************************************************************************
*/
ComponentResult QTAPI DataHOpenForRead (DHLR_FPARM2)
{
   void far *storageH = instanceStorage;
   DataHInstanceStoragePtr storage;
   char *fileName;

   storage = (DataHInstanceStoragePtr)GlobalLock((const void
                                                  near
*)LOWORD(storageH));
   if(storage == NULL)
      return(insufficientMemory);

   // must have a data reference
   if(storage->fileName == NULL)
   {
      GlobalUnlock((const void near *)LOWORD(storageH));
      return(dataNoDataRef);
   }
```

```
   // dereference the handle to the file name
   fileName = (char *)GlobalLock(storage->fileName);
   if(fileName == NULL)
   {
      GlobalUnlock((const void near *)LOWORD(storageH));
      return(invalidDataRef);
   }

   // open the file
   storage->fileRefNum = mmioOpen(fileName, NULL, MMIO_READ);
   if(storage->fileRefNum == NULL)
   {
      GlobalUnlock((const void near *)LOWORD(storageH));
      GlobalUnlock(storage->fileName);
      return(invalidDataRef);
   }

   // unlock handles
   GlobalUnlock((const void near *)LOWORD(storageH));
   GlobalUnlock(storage->fileName);

   // done
   if(storage->fileRefNum)
      return(noErr);
   else
      return(invalidDataRef);
}


/*
**************************************************************************
**
** Name: DataHCloseForRead()
**
** Description:
**
** Close data reference that has been opened for read access.
**
**************************************************************************
*/
ComponentResult QTAPI DataHCloseForRead (DHLR_FPARM2)
{
   void far *storageH = instanceStorage;
   DataHInstanceStoragePtr storage;

   // locate instance storage
   storage = (DataHInstanceStoragePtr)GlobalLock((const void
                                                  near
*)LOWORD(storageH));
   if(storage == NULL)
      return(insufficientMemory);

   // close the file
   if (storage->fileRefNum)
```

```
   {
      mmioClose(storage->fileRefNum, 0);
      storage->fileRefNum = 0;
   }
   else
      return(dataNotOpenForRead);

   // done
   return(noErr);
}


/*
**************************************************************************
**
** Name: DataHSetDataRef()
**
** Description:
**
** Set data reference for this component instance.  In QTW, the data
reference
** is the file path.  The input data reference is assumed to be a locked
** HGLOBAL.
**
**************************************************************************
*/
ComponentResult QTAPI DataHSetDataRef (DHLR_FPARM1
   Handle dataRef)
{
   char far *strIn;
   char *myStr;
   int len;
   HLOCAL mem;
   void far *storageH = instanceStorage;
   DataHInstanceStoragePtr storage;

   // locate instance storage
   storage = (DataHInstanceStoragePtr)GlobalLock((const void
                                                  near
*)LOWORD(storageH));
   if(storage == NULL)
      return(insufficientMemory);

   // release any existing allocated memory
   if(storage->fileName)
   {
      GlobalFree(storage->fileName);
      storage->fileName = NULL;
   }

   // deref the path
   strIn = (char far *)GlobalLock(dataRef);
   if(strIn == NULL)
   {
```

```
      GlobalUnlock((const void near *)LOWORD(storageH));
      return(invalidUserDataHandle);
   }
   len = _fstrlen(strIn);

   // allocate the memory
   mem = GlobalAlloc(GMEM_ZEROINIT, len);
   if(mem == NULL)
   {
      GlobalUnlock((const void near *)LOWORD(storageH));
      GlobalUnlock(dataRef);
      return(insufficientMemory);
   }

   // copy data
   myStr = (char *)GlobalLock(mem);
   while (*myStr++ = *strIn++)
      /* empty body */;
   GlobalUnlock(mem);

   // store handle
   storage->fileName = mem;

   // done
   GlobalUnlock((const void near *)LOWORD(storageH));
   return(noErr);
}


/*
**************************************************************************
**
** Name: DataHGetDataRef()
**
** Description:
**
** Return data reference for this component instance.
**
**************************************************************************
*/
ComponentResult QTAPI DataHGetDataRef (DHLR_FPARM1
   Handle *dataRef)
{
   char far *strOut;
   char *myStr;
   int len;
   HGLOBAL mem;
   void far *storageH = instanceStorage;
   DataHInstanceStoragePtr storage;

   // locate instance storage
   storage = (DataHInstanceStoragePtr)GlobalLock((const void
                                                  near
*)LOWORD(storageH));
```

```
   if(storage == NULL)
      return(insufficientMemory);

   // deref the path
   myStr = (char *)GlobalLock(storage->fileName);
   len = strlen(myStr);

   // allocate the memory
   mem = GlobalAlloc(0, len);
   if(mem == NULL)
   {
      GlobalUnlock(storage->fileName);
      GlobalUnlock((const void near *)LOWORD(storageH));
      return(insufficientMemory);
   }

   // copy data
   strOut = (char far *)GlobalLock(mem);
   while (*myStr++ = *strOut++)
      /* empty body */;
   GlobalUnlock(mem);

   // store handle
   *dataRef = mem;

   // done
   GlobalUnlock((const void near *)LOWORD(storageH));
   return(noErr);
}


/*
**************************************************************************
**
** Name: DataHCompareDataRef()
**
** Description:
**
** Compare provided data reference with the one established for this
** component instance.
**
**************************************************************************
*/


ComponentResult QTAPI DataHCompareDataRef (DHLR_FPARM1
   Handle dataRef,
   Boolean *equal)
{
   char far *inStr;
   char *myStr;
   int myLen, inLen;
   void far *storageH = instanceStorage;
```

```
   DataHInstanceStoragePtr storage;

   // locate instance storage
   storage = (DataHInstanceStoragePtr)GlobalLock((const void
                                                    near
*)LOWORD(storageH));
   if(storage == NULL)
      return(insufficientMemory);

   // deref the paths
   myStr = (char *)GlobalLock(storage->fileName);
   if(myStr == NULL)
   {
      GlobalUnlock((const void near *)LOWORD(storageH));
      return(insufficientMemory);
   }
   myLen = strlen(myStr);
   inStr = (char far *)GlobalLock(dataRef);
   if(myStr == NULL)
   {
      GlobalUnlock(storage->fileName);
      GlobalUnlock((const void near *)LOWORD(storageH));
      return(invalidUserDataHandle);
   }
   inLen = _fstrlen(inStr);

   // assume equal
   *equal = TRUE;

   // check lengths
   if(myLen != inLen)
   {
      *equal = FALSE;
   }
   else
   {
      // lengths are same, so check contents
      for(int i = 0; i < myLen; i++)
      {
         if(toupper(*myStr) != toupper(*inStr))
         {
            *equal = FALSE;
            break;
         }
         myStr++;
         inStr++;
      }
   }

   // done
   GlobalUnlock(storage->fileName);
   GlobalUnlock((const void near *)LOWORD(storageH));
   GlobalUnlock(dataRef);
   return(noErr);
```

```
}


/*
**********************************************************************
**
** Name: DataHTask()
**
** Description:
**
** Provides time slices for the data handler to perform background
operations.
**
**********************************************************************
*/
ComponentResult QTAPI DataHTask (DHLR_FPARM2)
{
   void far *storageH = instanceStorage;
   DataHInstanceStoragePtr storage;
   DataHReadRequestPtr request;

   // locate instance storage
   storage = (DataHInstanceStoragePtr)GlobalLock((const void
                                                  near
*)LOWORD(storageH));
   if(storage == NULL)
      return(insufficientMemory);

   // attempt to satisfy pending requests
   do
   {
      request = (DataHReadRequestPtr)wwList_GetHead(&storage-
>readRequestList);
      if(request)
      {
         if(_DataHDirectRead(storage, (char *)request-
>placeToPutDataPtr,
                             request->fileOffset, request->dataSize))
         {
            // do the callback
            if(request->completionRtn)
               (*request->completionRtn)(request->placeToPutDataPtr,
                  request->refCon, noErr);

            // remove this request.  The current list item will be
adjusted
            wwList_DelHead(&storage->readRequestList);
         }
         else
         {
            // move to the next item in the list
            wwList_GetNext(&storage->readRequestList);
         }
      }
```

```
      else
         request=(DataHReadRequestPtr)wwList_GetCurr(&storage-
>readRequestList);
   } while(request);

   // done
   GlobalUnlock((const void near *)LOWORD(storageH));
   return(noErr);
}


/*
**************************************************************************
**
** Name: DataHScheduleData()
**
** Description:
**
** Async or synchronous read operation.
**
**************************************************************************
*/
ComponentResult QTAPI DataHScheduleData (DHLR_FPARM1
   Ptr placeToPutDataPtr,
   long fileOffset,
   long dataSize,
   long refCon,
   DataHSchedulePtr scheduleRec,
   DHCompleteProc completionRtn)
{
   void far *storageH = instanceStorage;
   DataHInstanceStoragePtr storage;
   DataHReadRequestPtr request;

   // lock instance storage
   storage = (DataHInstanceStoragePtr)GlobalLock((const void
                                                  near
*)LOWORD(storageH));
   if(storage == NULL)
      return(insufficientMemory);

   // to read the data right now
   if(scheduleRec == NULL)
   {
      // use direct method to read from disk
      if(_DataHDirectRead(storage, (char *)placeToPutDataPtr,
fileOffset,
                          dataSize) == FALSE)
      {
         GlobalUnlock((const void near *)LOWORD(storageH));
         if(completionRtn)
            (*completionRtn)(placeToPutDataPtr, refCon, dataReadErr);
         return(dataReadErr);
      }
```

```
      // do the callback
      if(completionRtn)
         (*completionRtn)(placeToPutDataPtr, refCon, noErr);
   }
   else // the request is asynchronous
   {
      // allocate a new request structure
      request = (DataHReadRequestPtr)malloc(sizeof(DataHReadRequest));
      if(request == NULL)
      {
         GlobalUnlock((const void near *)LOWORD(storageH));
         if(completionRtn)
            (*completionRtn)(placeToPutDataPtr, refCon,
insufficientMemory);
         return(insufficientMemory);
      }

      // init request fields
      request->placeToPutDataPtr       = placeToPutDataPtr;
      request->fileOffset              = fileOffset;
      request->dataSize             = dataSize;
      request->refCon                  = refCon;
      request->completionRtn           = completionRtn;
      request->scheduleRec.timeNeededBy   = scheduleRec->timeNeededBy;
      request->scheduleRec.extendedID     = scheduleRec->extendedID;
      request->scheduleRec.extendedVers   = scheduleRec->extendedVers;
      request->scheduleRec.priority    = scheduleRec->priority;

      // place it in the request queue
      if(wwList_AddTail(&storage->readRequestList, (LISTDATA)request) ==
FALSE)
      {
         free(request);
         GlobalUnlock((const void near *)LOWORD(storageH));
         if(completionRtn)
            (*completionRtn)(placeToPutDataPtr, refCon,
insufficientMemory);
         return(insufficientMemory);
      }
   }

   // done
   GlobalUnlock((const void near *)LOWORD(storageH));
   return(noErr);
}


/*
*****************************************************************************
**
** Name: DataHFinishData()
**
** Description:
```

```
**
** Complete specified async read requests.
**
*****************************************************************************
*/
ComponentResult QTAPI DataHFinishData (DHLR_FPARM1
   Ptr placeToPutDataPtr,
   Boolean cancel )
{
   return(badComponentSelector);
}


/*
*****************************************************************************
**
** Name: DataHFlushCache()
**
** Description:
**
** Flush read caches.
**
*****************************************************************************
*/
ComponentResult QTAPI DataHFlushCache (DHLR_FPARM2)
{
   return(noErr);
}


/*
*****************************************************************************
**
** Name: DataHResolveDataRef()
**
** Description:
**
** Resolves a data reference.  No operation is performed, as a data
reference
** under QuickTime for Windows is a path name.
**
*****************************************************************************
*/
ComponentResult QTAPI DataHResolveDataRef (DHLR_FPARM1
   Handle dataRef,
   Boolean *wasChanged,
   Boolean userInterfaceAllowed)
{
   *wasChanged = FALSE;
   return(noErr);
}


/*
```

```
*****************************************************************************
**
** Name: DataHGetFileSize()
**
** Description:
**
** Return size of data reference.  The data reference must already be
open
** for this call to work.
**
*****************************************************************************
*/
ComponentResult QTAPI DataHGetFileSize (DHLR_FPARM1
   long *fileSize)
{
   void far *storageH = instanceStorage;
   DataHInstanceStoragePtr storage;
   long curr;

   // locate instance storage
   storage = (DataHInstanceStoragePtr)GlobalLock((const void
                                                  near
*)LOWORD(storageH));
   if(storage == NULL)
      return(insufficientMemory);

   // file must be open
   if (storage->fileRefNum == NULL)
   {
      GlobalUnlock((const void near *)LOWORD(storageH));
      return(dataNotOpenForRead);
   }

   // get the current file position and save it
   curr = mmioSeek(storage->fileRefNum, 0, SEEK_CUR);

   // get the size of the file by seeking to the EOF
   *fileSize = mmioSeek(storage->fileRefNum, 0, SEEK_END);
   if(*fileSize == -1)
   {
      *fileSize = 0;
      GlobalUnlock((const void near *)LOWORD(storageH));
      return(-1);
   }

   // reset the previous file position
   mmioSeek(storage->fileRefNum, curr, SEEK_SET);

   // done
   GlobalUnlock((const void near *)LOWORD(storageH));
   return(noErr);
}
```

```
/*
**************************************************************************
**
** Name: DataHCanUseDataRef()
**
** Description:
**
** Return flags indicating the ability for the data handler to access
the
** data reference.
**
** Currently only reading of files is supported.
**
**************************************************************************
*/
ComponentResult QTAPI DataHCanUseDataRef (DHLR_FPARM1
   Handle dataRef,
   DataHUseFlags *useFlags)
{
   *useFlags = kDataHCanRead;
   return(noErr);
}


/*
**************************************************************************
**
** Name: DataHGetVolumeList()
**
** Description:
**
** Return a list of volumes supported by this data handler.  Not
supported.
**
**************************************************************************
*/
ComponentResult QTAPI DataHGetVolumeList (DHLR_FPARM1
   DataHVolumeList *volumeList)
{
   return(badComponentSelector);
}


/*
**************************************************************************
********
**
** Name: DataHWrite()
**
** Description:
**
** Write data to data reference.  Not supported.
**
**************************************************************************
```

```
*/
ComponentResult QTAPI DataHWrite (DHLR_FPARM1
   Ptr data,
   long offset,
   long size,
   DHCompleteProc completion,
   long refcon)
{
   return(badComponentSelector);
}


/*
**************************************************************************
**
** Name: DataHPreextend()
**
** Description:
**
** Preextend the data reference.  Not supported.
**
**************************************************************************
*/
ComponentResult QTAPI DataHPreextend (DHLR_FPARM1
   long maxToAdd,
   long *spaceAdded)
{
   return(badComponentSelector);
}


/*
**************************************************************************
**
** Name: DataHSetFileSize()
**
** Description:
**
** Change file size of data reference.  Not supported.
**
**************************************************************************
*/
ComponentResult QTAPI DataHSetFileSize (DHLR_FPARM1
   long fileSize)
{
   return(badComponentSelector);
}


/*
**************************************************************************
**
** Name: DataHGetFreeSpace()
**
```

```
** Description:
**
** Return amount of free space on the device holding the data reference.
** Not supported.
**
*****************************************************************************
*/
ComponentResult QTAPI DataHGetFreeSpace (DHLR_FPARM1
   unsigned long *freeSize)
{
   void far *storageH = instanceStorage;
   DataHInstanceStoragePtr storage;
   char *fileName;
   int deviceIndex;
   _diskfree_t freeSpace;
   OSErr err;

   // locate instance storage
   storage = (DataHInstanceStoragePtr)GlobalLock((const void
                                                  near
*)LOWORD(storageH));
   if(storage == NULL)
      return(insufficientMemory);

   // must have a data reference
   if(storage->fileName == NULL)
   {
      GlobalUnlock((const void near *)LOWORD(storageH));
      return(dataNoDataRef);
   }

   // dereference the handle to the file name
   fileName = (char *)GlobalLock(storage->fileName);
   if(fileName == NULL)
   {
      GlobalUnlock((const void near *)LOWORD(storageH));
      return(invalidDataRef);
   }

   // get the index, use the drive letter.  If the caller didn't
   // specify a drive letter, then get the current drive.  The
   // index is 0 for A:, 1 for B:, ...
   if(fileName[1] != ':')
      deviceIndex = _getdrive();
   else
      deviceIndex = tolower(fileName[0])-'a';

   // get the free space on the device
   if(_dos_getdiskfree(deviceIndex, &freeSpace))
   {
      err = couldNotResolveDataRef;
   }
   else
   {
```

```
      // calculate the free space from the dos info returned
      *freeSize = freeSpace.avail_clusters *
freeSpace.sectors_per_cluster
                     * freeSpace.bytes_per_sector;
      err = noErr;
   }

   // unlock handles
   GlobalUnlock((const void near *)LOWORD(storageH));
   GlobalUnlock(storage->fileName);

   // done
   return(err);
}


/*
**********************************************************************
**
** Name: DataHCreateFile()
**
** Description:
**
** Create a file corresponding to the data reference.  Not supported.
**
**********************************************************************
*/
ComponentResult QTAPI DataHCreateFile (DHLR_FPARM1
   OSType creator,
   Boolean deleteExisting)
{
   return(badComponentSelector);
}


/*
**********************************************************************
**
** Name: DataHGetPreferredBlockSize()
**
** Description:
**
** Return the block size, in bytes, the data handler prefers to work
with.
**
**********************************************************************
*/
ComponentResult QTAPI DataHGetPreferredBlockSize (DHLR_FPARM1
   long *blockSize)
{
   // we are happiest with blocks of this size
   *blockSize = 512;
   return(noErr);
}
```

```
/*
**************************************************************************
**
** Name: DataHGetDeviceIndex()
**
** Description:
**
** Return a unique identifier for the device the data reference resides
on.
**
**************************************************************************
*/
ComponentResult QTAPI DataHGetDeviceIndex (DHLR_FPARM1
   long *deviceIndex)
{
   void far *storageH = instanceStorage;
   DataHInstanceStoragePtr storage;
   char *fileName;

   storage = (DataHInstanceStoragePtr)GlobalLock((const void
                                                  near
*)LOWORD(storageH));
   if(storage == NULL)
      return(insufficientMemory);

   // must have a data reference
   if(storage->fileName == NULL)
   {
      GlobalUnlock((const void near *)LOWORD(storageH));
      return(dataNoDataRef);
   }

   // dereference the handle to the file name
   fileName = (char *)GlobalLock(storage->fileName);
   if(fileName == NULL)
   {
      GlobalUnlock((const void near *)LOWORD(storageH));
      return(invalidDataRef);
   }

   // get the index, use the drive letter.  If the caller didn't
   // specify a drive letter, then get the current drive.  The
   // index is 0 for A:, 1 for B:, ...
   if(fileName[1] != ':')
      *deviceIndex = _getdrive();
   else
      *deviceIndex = (long)(tolower(fileName[0])-'a');

   // unlock handles
   GlobalUnlock((const void near *)LOWORD(storageH));
   GlobalUnlock(storage->fileName);
```

```
   // done
   return(noErr);
}


/*
**********************************************************************
**
** Name: DataHGetScheduleAheadTime()
**
** Description:
**
** Return schedule ahead time that the data handler prefers.  Currently
** an arbitrary value is returned.
**
**********************************************************************
*/
ComponentResult QTAPI DataHGetScheduleAheadTime (DHLR_FPARM1
   long *millisecs)
{
   // 2 seconds, arbitrary
   *millisecs = 2*1000;
   return(noErr);
}


/*
**********************************************************************
**
** Name: DataHPlaybackHints()
**
** Description:
**
** Provides hints about the data reference being accessed.  This
function
** may be called at any time, even during movie playback if the user has
** made edits to the movie.
**
**********************************************************************
*/
ComponentResult QTAPI DataHPlaybackHints (DHLR_FPARM1
   long flags,
   unsigned long minFileOffset,
   unsigned long maxFileOffset,
   long bytesPerSecond)
{
   void far *storageH = instanceStorage;
   DataHInstanceStoragePtr storage;

   // locate instance storage
   storage = (DataHInstanceStoragePtr)GlobalLock((const void
                                                  near
*)LOWORD(storageH));
   if(storage == NULL)
```

```
      return(insufficientMemory);

   // store playback hints
   storage->minFileOffset = minFileOffset;
   storage->maxFileOffset = maxFileOffset;
   storage->bytesPerSecond = bytesPerSecond;

   // done
   GlobalUnlock((const void near *)LOWORD(storageH));
   return(noErr);
}


/*
**************************************************************************
**
** Name: DataHSetOSFileReference()
**
** Description:
**
** Set the file reference directly to an already open file.  This call
** exists because NewMovieFromDataFork() is only given an HFILE to work
with,
** and MS-Windows can't backup to the filename from just the HFILE.
**
**************************************************************************
*/
ComponentResult QTAPI DataHSetOSFileReference (DHLR_FPARM1
   long fileRef,
   long filePerms)
{
   void far *storageH = instanceStorage;
   DataHInstanceStoragePtr storage;

   // locate instance storage
   storage = (DataHInstanceStoragePtr)GlobalLock((const void
                                                  near
*)LOWORD(storageH));
   if(storage == NULL)
      return(insufficientMemory);

   // make sure we are not already open via the regular data reference
   if(storage->fileRefNum)
      return(invalidDataRef);

   // assign the file reference
   storage->hfileRefNum = (HFILE)fileRef;
   storage->hfilePerms = filePerms;

   // done
   GlobalUnlock((const void near *)LOWORD(storageH));
   return(noErr);
}
```

```
/*
**********************************************************************
**
** Name: DataHGetOSFileReference()
**
** Description:
**
** Returns the file reference set by SetOSFileReference();
**
**********************************************************************
*/
ComponentResult QTAPI DataHGetOSFileReference (DHLR_FPARM1
   long *fileRef,
   long *filePerms)
{
   void far *storageH = instanceStorage;
   DataHInstanceStoragePtr storage;

   // locate instance storage
   storage = (DataHInstanceStoragePtr)GlobalLock((const void
                                                  near
*)LOWORD(storageH));
   if(storage == NULL)
      return(insufficientMemory);

   // make sure we are not already open via the regular data reference
   if(storage->fileRefNum)
      return(invalidDataRef);

   // copy the file reference
   *fileRef = storage->hfileRefNum;
   *filePerms = storage->hfilePerms;

   // done
   GlobalUnlock((const void near *)LOWORD(storageH));
   return(noErr);
}


/*
**********************************************************************
**
** Name: _DataHDirectRead()
**
** Description:
**
** Directly calls mmioRead() to read data from a file.  This function
does
** not alter the current file position (it is preserved).  This function
** should be used only as a last resort, i.e. if the TdmRead() function
** cannot return the data.
**
**********************************************************************
```

```
*/
BOOL _DataHDirectRead(
   DataHInstanceStorage *storage,   // storage for this instance
   char *pDestBuf,      // pointer to destination buffer
   long fileOffset,  // file offset to begin read at
   long size)        // # of bytes to read
{
   BOOL result = TRUE;
   UINT xfer;
   long nread;
   unsigned long mysize;

   if(storage->fileRefNum) {
      // move to the new location
      mmioSeek(storage->fileRefNum, fileOffset, SEEK_SET);

      // read the data
      nread = mmioRead(storage->fileRefNum, pDestBuf, size);
      if(nread < size)
         result = FALSE;
   }
   else if(storage->hfileRefNum) {
      // the file reference was set via SetOSFileReference, so read
      // in the data in 64k chunks
      mysize = size;
      while(mysize) {
         xfer = (UINT)(mysize < 65536 ? mysize : 65535);
         _lread(storage->hfileRefNum, pDestBuf, xfer);
         pDestBuf += xfer;
         mysize -= xfer;
      }
   }
   else {
      result = FALSE;
   }

   // done
   return(result);
}
```

REFERENCE TO DATA HANDLER COMPONENTS

This section describes the functions your data handler component may support. Some of these functions are optional—your component should support only those functions that are appropriate to it.

Functions

This section describes the functions that may be supported by data handler components, and is divided into the following topics:

- "Selecting a Data Handler" discusses the functions that allow client programs, such as the Movie Toolbox, to select an appropriate data handler for a data reference.
- "Working With Data References" describes the functions that allow client programs to manage a data handler's current data reference.
- "Reading Movie Data" tells you about the functions that allow client programs to retrieve data from a data handler.
- "Writing Movie Data" tells you about the functions that allow client programs to store data using a data handler.
- "Managing Data Handler Components" provides information about the functions that allow client programs to manage their interactions with data handler components.
- "Completion Function" discusses the interface that must be provided by a client program's data-handler completion function.

Selecting a Data Handler

In order to client programs to choose the best data handler component for a data reference, Apple has defined some functions that allow applications to interrogate a data handler's capabilities.

The `DataHGetVolumeList` function allows an application to obtain a list of the volumes your data handler can support. The `DataHCanUseDataRef` function allows your data handler to examine a specific data reference and indicate its ability to work with the associated container. The `DataHGetDeviceIndex` function allows applications to determine whether different data references identify containers that reside on the same device.

By way of illustration, the Movie Toolbox uses the `DataHGetVolumeList` and `DataHCanUseDataRef` functions as follows. During startup, and whenever a new volume is mounted, the Movie Toolbox calls each data handler's `DataHGetVolumeList` function in order to obtain information about each handler's general capabilities. So, the Movie Toolbox calls each component's `OpenComponent`, `DataHGetVolumeList` , and `CloseComponent` functions.

Whenever an application opens a movie, the Movie Toolbox selects the best data handler for the movie's container. This may involve calling each appropriate data handler's `DataHCanUseDataRef` function (in some cases, a data handler may indicate that it does not need to examine a data reference before accessing it—see the discussion of the `DataHGetVolumeList` function for more information). For each data handler that can support the data reference (that is, has the correct component subtype value) and needs to be interrogated, the Movie Toolbox calls the component's `OpenComponent`, `DataHCanUseDataRef` , and `CloseComponent` functions. Based on the resulting information, the Movie Toolbox selects the best data handler for the application.

DataHGetVolumeList

In response to the `DataHGetVolumeList` function, your data handler component returns a list of the volumes your component can access, along with flags indicating your component's capabilities for each volume.

```
pascal ComponentResult DataHGetVolumeList (DataHandler dh,
                                DataHVolumeList *volumeList);
```

`dh`	Identifies the calling program's connection to your data handler component.
`volumeList`	Contains a pointer to a field that your data handler component uses to return a handle to a volume list. Your component constructs the volume list by allocating a handle and filling it with a series of `DataHVolumeListRecord` structures (one structure for each volume your component can access). This structure is described later in this section.

DESCRIPTION

In order to reduce the delay that may result from choosing an appropriate data handler for a volume, the Movie Toolbox maintains a list of data handlers and the volumes they support. The Movie Toolbox uses the `DataHGetVolumeList` function to build that list.

When your component receives this function, it should scan the available volumes and create a series of `DataHVolumeListRecord` structures—one structure for each volume your component can access. This structure is defined as follows:

```
typedef struct DataHVolumeListRecord {
            short       vRefNum;    /* reference number */
```

```
long        flags;      /* capability flags */
} DataHVolumeListRecord, *DataHVolumeListPtr,
**DataHVolumeList;
```

`vRefNum` Contains the volume reference number assigned to the volume.

`flags` Indicates the level of support your data handler can provide for this volume. These flags are similar to those defined for the `DataHCanUseDataRef` function, though there is one additional flag. Your component should set every appropriate flag to 1 (set unused flags to 0).

`kDataHCanRead` Indicates that your data handler can read from the volume.

`kDataHSpecialRead` Indicates that your data handler can read from the volume using a specialized method. For example, your data handler might support access to networked multimedia servers using a special protocol. In that case, your component would set this flag to 1 whenever the volume resides on a supported server.

`kDataHSpecialReadFile` Reserved for use by Apple.

`kDataHCanWrite` Indicates that your data handler can write data to the volume. In particular, use this flag to indicate that your data handler's `DataHPutData` function will work with this volume.

`kDataHSpecialWrite` Indicates that your data handler can write to the volume using a specialized method. As with the `kDataHSpecialRead` flag, your data handler would use this flag to indicate that your component can access the volume using specialized support (for example, special network protocols).

`kDataHCanStreamingWrite`
Indicates that your data handler can support the special write functions for capturing movie data when writing to this volume. These functions are described later in this chapter, in “Writing Movie Data.”

`kDataHMustCheckDataRef`
Instructs the calling program that your component must check each data reference before it can accurately report its capabilities. If you set this flag to 1, the Movie Toolbox will call your component’s `DataHCanUseDataRef` function before it assigns a container to your data handler. Note, however, that this may slow the data handler selection process somewhat.

Your data handler may use any facilities necessary to determine whether it can access the volume, including opening a container on the volume. Your component should set to 1 as many of the capability flags as are appropriate for each volume. Do not include records for volumes your handler cannot support.

For example, if your component supports networked multimedia servers using a special set of protocols, your data handler should set the `kDataHCanRead` and `kDataHCanSpecialRead` flags to 1 for any volume that is on that server. In addition, if your component can write to a volume on the server, set the `kDataHCanWrite` and `kDataHCanSpecialWrite` flags to 1 (perhaps along with `kDataHCanStreamingWrite`). However, your component should create entries only for those volumes that support your protocols.

It is the calling program’s responsibility to dispose of the handle returned by your component.

The Movie Toolbox tracks mounting and unmounting removable volumes, and keeps its volume list current. As a result, the Movie Toolbox may call your component’s `DataHGetVolumeList` function whenever a removable volume is mounted.

If your data handler does not process data that is stored in file system volumes, you need not support this function.

ERROR CODES

Memory Manager errors

DataHCanUseDataRef

The `DataHCanUseDataRef` function allows your data handler to report whether it can access the data associated with a specified data reference.

```
pascal ComponentResult DataHCanUseDataRef (DataHandler dh,
                                Handle dataRef,
                                long *useFlags);
```

`dh` Identifies the calling program's connection to your data handler component.

`dataRef` Specifies the data reference. This parameter contains a handle to the information that identifies the container in question.

`useFlags` Contains a pointer to a field that your data handler component uses to indicate its ability to access the container identified by the `dataRef` parameter. Your data handler may use the following flags (set all flags that are appropriate to 1; set unused flags to 0):

`kDataHCanRead` Indicates that your data handler can read from the container.

`kDataHSpecialRead` Indicates that your data handler can read from the container using a specialized method. For example, your data handler might support access to networked multimedia servers using a special protocol. In that case, your component would set this flag to 1 whenever the data reference identifies a container on a supported server.

`kDataHSpecialReadFile` Indicates that your data handler can read from the container using a specialized method that is particular to the type of container in question. For example, your data handler may use a different method for some types of containers (say, a Hypercard stack).

This flag represents a special case of the `kDataHSpecialRead` flag. That is, this flag is appropriate only if you have also set `kDataHSpecialRead` to 1.

`kDataHCanWrite` Indicates that your data handler can write data to the container. In particular, use this flag to indicate that your data handler's `DataHPutData` function will work with this data reference.

`kDataHSpecialWrite`
Indicates that your data handler can write to the container using a specialized method. As with the `kDataHSpecialRead` flag, your data handler would use this flag to indicate that the data reference identifies a container which your component can access using specialized support (for example, special network protocols).

`kDataHCanStreamingWrite`
Indicates that your data handler can support the special write functions for capturing movie data when writing to this container. These functions are described later in this chapter, in "Writing Movie Data."

If your data handler cannot access the container, set the field to 0.

DESCRIPTION

Apple's standard data handler sets both the `kDataHCanRead` and `kDataHCanWrite` flags to 1 for any data reference it receives, indicating that it can read from and write to any volume.

Your component should set to 1 as many of the capability flags as are appropriate for the specified data reference. Conversely, be sure to set the flags to 0 if your component cannot support the container. For example, if your component supports networked multimedia servers using a special set of protocols, your data handler should set the `kDataHCanRead` and `kDataHCanSpecialRead` flags to 1 for any container that is on that server. In addition, if your component can write to the server, set the `kDataHCanWrite` and `kDataHCanSpecialWrite` flags to 1 (perhaps along with `kDataHCanStreamingWrite`). However, your component should set the flags field to 0 for any container that is not on a server that supports your protocols.

Your data handler may use any facilities necessary to determine whether it can access the container. Bear in mind, though, that your component should try to be as quick about this determination as possible, in order to minimize the chance that the delay will be noticed by the user.

SEE ALSO

The Movie Toolbox calls your component's `DataHGetVolumeList` function to retrieve your data handler's capabilities for an entire volume.

DataHGetDeviceIndex

In response to the `DataHGetDeviceIndex` function, your data handler component returns a value that identifies the device on which a data reference resides.

```
pascal ComponentResult DataHGetDeviceIndex (DataHandler dh,
                                long *deviceIndex);
```

`dh`	Identifies the calling program's connection to your data handler component.
`deviceIndex`	Contains a pointer to a field that your data handler component uses to return a device identifier value.

DESCRIPTION

Some client programs may need to account for the fact that two or more data references reside on the same device. For instance, this may affect storage-allocation requirements. This function allows such client programs to obtain this information from your data handler.

Your component may use any identifier value that is appropriate (as an example, Apple's HFS data handler uses the volume reference number). The client program should do nothing with the value other than compare it with other identifiers returned by your data handler component.

Working With Data References

All data handler components use data references to identify and locate a movie's container. Different types of containers may require different types of data references. For example, a reference to a memory-based movie may be a handle, while a reference to a file-based movie may be an alias.

Client programs can correlate data references with data handlers by matching the component's subtype value with the data reference type—the subtype value indicates the type of data reference the component supports. All data handlers with the same subtype value must support the same data reference type. To continue the previous example, Apple's memory-based data handler for the Macintosh uses handles (and has a subtype value of `'hndl'`), while the HFS data handler uses Alias Manager aliases (its subtype value is `'alis'`).

The `DataHSetDataRef` and `DataHGetDataRef` functions allow applications to assign your data handler's current data reference. The `DataHCompareDataRef` function asks your component to compare a data reference against the current data reference and indicate whether the references are equivalent (that is, refer to the same container). The `DataHResolveDataRef` permits your component to locate a data reference's container.

The `DataHSetOSFileRef` and `DataHGetOSFileRef` functions provide an alternative, system-specific mechanism for assigning your data handler's current data reference.

DataHSetDataRef

The `DataHSetDataRef` function assigns a data reference to your data handler component.

```
pascal ComponentResult DataHSetDataRef (DataHandler dh,
                                        Handle dataRef);
```

`dh`	Identifies the calling program's connection to your data handler component.
`dataRef`	Specifies the data reference. This parameter contains a handle to the information that identifies the container in question. Your component must make a copy of this handle.

DESCRIPTION

Note that the type of data reference always corresponds to the type that your component supports, and that you specify in the component subtype value of your data handler. As a result, the client program does not provide a data reference type value (unlike the Movie Toolbox's data reference functions).

The client program is responsible for disposing of the handle. Consequently, your component must make a copy of the data reference handle.

ERROR CODES

Memory Manager errors

DataHGetDataRef

The `DataHGetDataRef` function retrieves your component's current data reference.

```
pascal ComponentResult DataHGetDataRef (DataHandler dh,
                                        Handle *dataRef);
```

`dh`	Identifies the calling program's connection to your data handler component.
`dataRef`	Contains a pointer to a data reference handle. Your component should make a copy of its current data reference in a handle and return that handle in this field. The client program is responsible for disposing of that handle.

ERROR CODES

Memory Manager errors

DataHCompareDataRef

Your component compares a supplied data reference against its current data reference and returns a Boolean value indicating whether the data references are equivalent (that is, the two data references identify the same container).

```
pascal ComponentResult DataHCompareDataRef (DataHandler dh,
                                Handle dataRef, Boolean
                                *equal);
```

`dh`	Identifies the calling program's connection to your data handler component.
`dataRef`	Specifies the data reference to be compared to your component's current data reference.
`equal`	Contains a pointer to a Boolean. Your component should set that Boolean to `true` if the two data references identify the same container. Otherwise, set the Boolean to `false`.

DESCRIPTION

Note that your component cannot simply compare the bits in the two data references. For example, two completely different aliases may refer to the same HFS file. Consequently, you need to completely resolve the data reference in order to determine the file identified by the reference.

DataHResolveDataRef

The `DataHResolveDataRef` function instructs your data handler component to locate the container associated with a given data reference.

```
pascal ComponentResult DataHResolveDataRef (DataHandler dh,
                                Handle theDataRef,
```

```
                                    Boolean *wasChanged,
                                    Boolean userInterfaceAllowed);
```

`dh` Identifies the calling program's connection to your data handler component.

`theDataRef` Specifies the data reference to be resolved.

`wasChanged` Contains a pointer to a Boolean. Your component should set that Boolean to `true` if, in locating the container, your data handler updates any information in the data reference.

`userInterfaceAllowed`
Indicates whether your component may interact with the user when locating the container. If this parameter is set to `true`, your component may ask the user to help locate the container (for instance, by presenting a Find File dialog box).

DESCRIPTION

This function is, essentially, equivalent to the Alias Manager's `ResolveAlias` function. The client program asks your component to locate the container that is associated with a given data reference. If your component determines that the data reference needs to be updated with more accurate location information, it should put the new information in the supplied data reference (and set the Boolean referred to by the `wasChanged` parameter to `true`).

Client programs may call your data handler's `DataHResolveDataRef` function at any time. Typically, however, the Movie Toolbox uses this function as part of its strategy for opening and reading a movie container. As such, you can expect that the supplied data reference will identify a container that your component can support.

DataHSetOSFileRef

The `DataHSetOSFileRef` function assigns a movie container to your data handler component. Applications may use this function instead of calling the `DataHSetDataRef` function in cases where the applications have already opened the container.

```
pascal ComponentResult DataHSetOSFileRef (DataHandler dh,
                                    long ref, long flags);
```

`dh` Identifies the calling program's connection to your data handler component.

`ref`	Specifies the container. This parameter contains an operating system-specific file-access token. For example, on the Macintosh an application would supply the file reference it obtained by calling the `FSOpenFile` function. Under Windows, this parameter would contain an `HFILE` value obtained from the `OpenFile` function.
`flags`	Specifies access flags for the container. This parameter contains the access flags the application used when opening the container. Again, these are operating system-specific.

DESCRIPTION

This function provides an alternative mechanism for assigning your data handler's current container. In some cases, an application may have created or opened a movie container prior to assigning the container to your handler. In such cases, the application may choose to provide its access token to your data handler, rather than using the `DataHSetDataRef` function to assign a data reference. The application must have opened the file before calling this function.

Note that your data handler must implement this function in a system-specific manner, and must verify that the access token is valid.

Applications must still call your handlers `DataHOpenForRead` or `DataHOpenForWrite` functions, as appropriate, before using your data handler to access the container.

ERROR CODES

`invalidDataRef`	–2012	Application already set a data reference
`memFullErr`	–108	Insufficient memory for operation

DataHGetOSFileRef

The `DataHGetOSFileRef` function retrieves your component's container access token, if it was assigned using the `DataHSetOSFileRef` function.

```
pascal ComponentResult DataHGetOSFileRef (DataHandler dh,
                                long *ref, long *flags);
```

`dh`	Identifies the calling program's connection to your data handler component.
`ref`	Contains a pointer to a long. Your component should return the container access token that the application provided when it called your `DataHSetOSFileRef` function.

`flags`	Contains a pointer to a long. Your component should return the access flags that the application provided when it called your `DataHSetOSFileRef` function.

ERROR CODES

`invalidDataRef`	–2012	Application already set a data reference
`memFullErr`	–108	Insufficient memory for operation

Reading Movie Data

Data handler components provide two basic read facilities. The `DataHGetData` function is a fully synchronous read operation, while the `DataHScheduleData` function is asynchronous. Applications provide scheduling information when they call your component's `DataHScheduleData` function. When your component processes the queued request, it calls the application's data-handler completion function (see "Completion Function," later in this chapter, for more information). By calling your component's `DataHFinishData` function, applications can force your component to process queued read requests. Applications may call your component's `DataHGetScheduleAheadTime` function in order to determine how far in advance your component prefers to get read requests.

Before any application can read data from a data reference, it must open read access to that reference by calling your component's `DataHOpenForRead` function. The `DataHCloseForRead` function closes that read access path.

DataHOpenForRead

Your component opens its current data reference for read-only access.

```
pascal ComponentResult DataHOpenForRead (DataHandler dh);
```

`dh`	Identifies the calling program's connection to your data handler component.

DESCRIPTION

After setting your component's current data reference by calling the `DataHSetDataRef` function, client programs call the `DataHOpenForRead` function in order to start reading from the data reference. Your component should open the data reference for read-only access. If the data reference is already open or cannot be opened, return an appropriate error code.

Note that the Movie Toolbox may try to read data from a data reference without calling your component's `DataHOpenForRead` function. If this happens, your component should open the data reference for read-only access, respond to the read request, and then leave the data reference open in anticipation of later read requests.

DataHCloseForRead

Your component closes read-only access to its data reference.

```
pascal ComponentResult DataHCloseForRead (DataHandler dh);
```

`dh`	Identifies the calling program's connection to your data handler component.

DESCRIPTION

Note that a client program may close its connection to your component (by calling the Component Manager's `CloseComponent` function) without closing the read path. If this happens, your component should close the data reference before closing the connection.

ERROR CODES

`dataNotOpenForRead`	–2042	Data reference not open for read
`dataAlreadyClosed`	–2045	This reference already closed

DataHGetData

Your component reads data from its current data reference. This is a synchronous read operation.

```
pascal ComponentResult DataHGetData (DataHandler dh, Handle
                                     h, long hOffset, long offset,
                                     long size);
```

`dh`	Identifies the calling program's connection to your data handler component.
`h`	Specifies the handle to receive the data.
`hOffset`	Identifies the offset into the handle where your component should return the data.
`offset`	Specifies the offset in the data reference from which your component is to read.
`size`	Specifies the number of bytes to read.

DESCRIPTION

The `DataHGetData` function provides a high-level read interface. This is a synchronous read operation; that is, the client program's execution is blocked until your component returns control from this function. As a result, most time-critical clients use the `DataHScheduleData` function to read data.

Note that the Movie Toolbox may try to read data from a data reference without calling your component's `DataHOpenForRead` function. If this happens, your component should open the data reference for read-only access, respond to the read request, and then leave the data reference open in anticipation of later read requests.

SEE ALSO

Client programs can force your component to invalidate any cached data by calling your component's `DataHFlushCache` function.

DataHScheduleData

Your component reads data from its current data reference. This can be a synchronous read operation or an asynchronous read operation.

```
pascal ComponentResult DataHScheduleData (DataHandler dh,
                                Ptr placeToPutDataPtr,
                                long fileOffset, long
                                dataSize, long refCon,
                                DataHSchedulePtr scheduleRec,
                                DHCompleteProc completionRtn);
```

`dh`	Identifies the calling program's connection to your data handler component.
`placeToPutDataPtr`	Specifies the location in memory that is to receive the data.
`fileOffset`	Specifies the offset in the data reference from which your component is to read.
`dataSize`	Specifies the number of bytes to read.
`refCon`	Contains a reference constant that your data handler component should provide to the data-handler completion function specified with the `completionRtn` parameter.

`scheduleRec` Contains a pointer to a schedule record. If this parameter is set to `nil`, then the client program is requesting a synchronous read operation (that is, your data handler must return the data before returning control to the client program).

If this parameter is not set to `nil`, it must contain the location of a schedule record that has timing information for an asynchronous read request. Your data handler should return control to the client program immediately, and then call the client's data-handler completion function when the data is ready. The schedule record is discussed later in this section.

`completionRtn` Contains a pointer to a data-handler completion function. When your data handler finishes with the client program's read request, your component must call this routine. Be sure to call this routine even if the request fails. Your component should pass the reference constant that the client program provided with the `refCon` parameter.

The client program must provide a completion routine for all asynchronous read requests (that is, all requests that include a valid schedule record). For synchronous requests, client programs should set this parameter to `nil`. However, if the function is provided, your handler must call it, even after synchronous requests.

DESCRIPTION

The `DataHScheduleData` function provides both a synchronous and an asynchronous read interface. Synchronous read operations work like the `DataHGetData` function—the data handler component returns control to the client program only after it has serviced the read request. Asynchronous read operations allow client programs to schedule read requests in the context of a specified QuickTime time base. Your data handler queues the request and immediately returns control to the calling program. After your component actually reads the data, it calls the client program's data-handler completion function.

If your component cannot satisfy the request (for example, the request requires data more quickly than you can deliver it), your component should reject the request immediately, rather than queuing the request and then calling the client's data-handler completion function.

The client program provides scheduling information for scheduled reads in a schedule record. This structure is defined as follows:

```
typedef struct DataHScheduleRecord {
            TimeRecord  timeNeededBy;     /* schedule info
            */
```

```
long        extendedID;       /* type of data */
long        extendedVers;     /* reserved */
Fixed       priority;         /* priority */
} DataHScheduleRecord, *DataHSchedulePtr;
```

`timeNeededBy` Specifies the time at which your data handler must deliver the requested data to the calling program. This time value is relative to the time base that is contained in this time record.

During pre-roll operations, the Movie Toolbox may use special values in certain time record fields. The time record fields in question are the `scale` and `value` fields. By correctly interpreting the values of these fields, your data handler can queue up the pre-roll read requests in the most efficient way for its device.

There are two types of pre-roll read operations. The first type is a required read; that is, the Movie Toolbox requires that the read operation be satisfied before the movie starts playing. The second type is an optional read. If your data handler can satisfy the read operation as part of the pre-roll operation, it should do so. Otherwise, your data handler may satisfy the request at a specified time while the movie is playing.

The Movie Toolbox indicates that a pre-roll read request is required by setting the `scale` field of the time record to –1. This literally means that the request is scheduled for a time that is infinitely far into the future. Your data handler should collect all such read requests, order them most efficiently for your device, and process them when the Movie Toolbox calls your component's `DataHFinishData` function.

For optional pre-roll read requests, the Movie Toolbox sets the `scale` field properly, but negates the contents of the `value` field. Your data handler has the option of delivering the data for this request with the required data, if that can be done efficiently. Otherwise, your data handler may deliver the data at its schedule time. You determine the scheduled time by negating the contents of the `value` field (that is, multiplying by –1).

For more information about pre-roll operations, see "Retrieving Movie Data," earlier in this chapter.

`extendedID` Indicates the type of data that follows in the remainder of the record. The following values are valid:

`kDataHExtendedSchedule`
The remainder of the record contains extended scheduling information.

If the `extendedID` field is set to `kDataHExtendedSchedule`, the remainder of the schedule record is defined as follows:

`extendedVers` Reserved; this field should always be set to 0.

`priority` Indicates the relative importance of the data request. Client programs assign a value of 100.0 to data requests the must be delivered. Lower values indicate relatively less critical data. If your data handler must accommodate bandwidth limitations when delivering data, your component may use this value as an indication of which requests can be dropped with the least impact on the client program.

As an example, consider using priorities in a frame-differenced movie. Key frames might have priority values of 100.0, indicating that they are essential to proper playback. As you move through the frames following a key frame, each successive frame might have a lower priority value. Once you drop a frame, you must drop all successive frames of equal or lower priority until you reach another key frame, because each of these frames would rely on the dropped one for some image data.

Note that the Movie Toolbox may try to read data from a data reference without calling your component's `DataHOpenForRead` function. If this happens, your component should open the data reference for read-only access, respond to the read request, and then leave the data reference open in anticipation of later read requests.

SEE ALSO

Client programs can force your component to invalidate any cached data by calling your component's `DataHFlushCache` function.

DataHFinishData

The `DataHFinishData` function instructs your data handler component to complete or cancel one or more queued read requests. The client program would have issued those read requests by calling your component's `DataHScheduleData` function.

```
pascal ComponentResult DataHFinishData (DataHandler dh,
                                        Ptr placeToPutDataPtr,
                                        Boolean cancel);
```

`dh`	Identifies the calling program's connection to your data handler component.
`placeToPutDataPtr`	Specifies the location in memory that is to receive the data. The value of this parameter identifies the specific read request to be completed. If this parameter is set to `nil`, the call affects all pending read requests.
`cancel`	Indicates whether the calling program wants to cancel the outstanding request. If this parameter is set to `true`, your data handler should cancel the request (or requests) identified by the `placeToPutDataPtr` parameter.

DESCRIPTION

Client programs use the `DataHFinishData` function either to cancel outstanding read requests or to demand that the requests be serviced immediately. Pre-roll operations are a special case of the immediate service request. The client program will have queued one or more read requests with their scheduled time of delivery set infinitely far into the future. Your data handler queues those requests until the client program calls the `DataHFinishData` function demanding that all outstanding read requests be satisfied immediately.

Note that your component must call the client program's data-handler completion function for each queued request, even though the client program called the `DataHFinishData` function. Be sure to call the completion function for both canceled and completed read requests.

SEE ALSO

Client programs queue read requests by calling your component's `DataHScheduleData` function.

DataHGetScheduleAheadTime

The `DataHGetScheduleAheadTime` function allows your data-handler component to report how far in advance it prefers clients to issue read requests.

```
pascal ComponentResult DataHGetScheduleAheadTime
                                        (DataHandler dh,
                                        long *millisecs);
```

`dh`	Identifies the calling program's connection to your data handler component.
`millisecs`	Contains a pointer to a long. Your component should set this field with a value indicating the number of milliseconds you prefer to receive read requests in advance of the time when the data must be delivered.

DESCRIPTION

This function allows your data handler to tell the client program how far in advance it should schedule its read requests. By default, the Movie Toolbox issues scheduled read requests between 1 and 2 seconds before it needs the data from those requests. For some data handlers, however, this may not be enough time. For example, some data handlers may have to accommodate network delays when processing read requests. Client programs that call this function may try to respect your component's preference.

Note, however, that not all client programs will call this function. Further, some clients may not be able to accommodate your preferred time in all cases, even if they have asked for your component's preference. As a result, your component should have a strategy for handling requests that do not provide enough advanced scheduling time. For example, if your component receives a `DataHScheduleData` request that it cannot satisfy, it can fail the request with an appropriate error code.

SEE ALSO

Client programs queue read requests by calling your component's `DataHScheduleData` function.

Writing Movie Data

As with reading movie data, data handlers provide two distinct write facilities. The `DataHPutData` function is a simple synchronous interface that allows applications to append data to the end of a container.

The `DataHWrite` function is a more-capable, asynchronous write function that is suitable for movie capture operations. As is the case with the `DataHScheduleData` function, your component calls the application's data-handler completion function when you are done with the write request.

There are several other helper functions that allow applications to prepare your data handler for a movie capture operation. The `DataHCreateFile` function asks your component to create a new container. The `DataHSetFileSize` and `DataHGetFileSize` functions work with a container's size, in bytes. The `DataHGetFreeSpace` function allows applications to determine when to make a container larger. The `DataHPreextend` function asks your component to make a container larger. Applications may call your component's `DataHGetPreferredBlockSize` function in order to determine how best to interact with your data handler.

Before writing data to a data reference, applications must call your component's `DataHOpenForWrite` function to open a write path to the container. The `DataHCloseForWrite` function closes that write path.

Note that some data handlers may not support write operations. For example, some shared devices, such as a CD-ROM "jukebox", may be read-only devices. As a result, it is very important that your data handler correctly report its write capabilities to client programs. See "Selecting a Data Handler," earlier in this chapter, for information about the functions that client programs use to interrogate your data handler.

DataHOpenForWrite

Your component opens its current data reference for write-only access.

```
pascal ComponentResult DataHOpenForWrite (DataHandler dh);
```

`dh` Identifies the calling program's connection to your data handler component.

DESCRIPTION

After setting your component's current data reference by calling the `DataHSetDataRef` function, client programs call the `DataHOpenForWrite` function in order to start writing to the data reference. Your component should open the data reference for write-only access. If the data reference is already open or cannot be opened, return an appropriate error code.

ERROR CODES

`dataAlreadyOpenForWrite`	–2044	Data reference already open for write

DataHCloseForWrite

Your component closes write-only access to its data reference.

```
pascal ComponentResult DataHCloseForWrite (DataHandler dh);
```

`dh` Identifies the calling program's connection to your data handler component.

DESCRIPTION

Note that a client program may close its connection to your component (by calling the Component Manager's `CloseComponent` function) without closing the write path. If this happens, your component should close the data reference before closing the connection.

ERROR CODES

`dataNotOpenForWrite`	–2043	Data reference not open for write
`dataAlreadyClosed`	–2045	This reference already closed

DataHPutData

Your component writes data to its current data reference. This is a synchronous write operation that appends data to the end of the current data reference.

```
pascal ComponentResult DataHPutData (DataHandler dh, Handle
                                     h, long hOffset, long *offset,
                                     long size);
```

`dh`	Identifies the calling program's connection to your data handler component.
`h`	Specifies the handle that contains the data to be written to the data reference.
`hOffset`	Identifies the offset into the handle `h` to the data to be written.
`offset`	Contains a pointer to a long. Your component returns the offset in the data reference at which your component wrote the data.
`size`	Specifies the number of bytes to write.

DESCRIPTION

The `DataHPutData` function provides a high-level write interface. This is a synchronous write operation that only appends data to the end of the current data reference. That is, the client program's execution is blocked until your component returns control from this function, and the client cannot control where the data is written. As a result, most movie-capture clients (for example, Apple's sequence grabber component) use the `DataHWrite` function to write data when creating movies.

ERROR CODES

`dataNotOpenForWrite`	–2043	Data reference not open for write

SEE ALSO

Client programs can force your component to write any cached data by calling your component's `DataHFlushData` function.

DataHWrite

Your component writes data to its current data reference. This can be a synchronous write operation or an asynchronous operation, and can write data to any location in the container.

```
pascal ComponentResult DataHWrite (DataHandler dh, Ptr data,
                                   long offset, long size,
                                   DHCompleteProc completion,
                                   long refCon);
```

`dh` Identifies the calling program's connection to your data handler component.

`data` Specifies a pointer to the data to be written. Client programs should lock the memory area holding this data, allowing your component's `DataHWrite` function to move memory.

`offset` Specifies the offset (in bytes) to the location in the current data reference at which to write the data.

`size` Specifies the number of bytes to write.

`completion` Contains a pointer to a data-handler completion function. When your data handler finishes with the client program's write request, your component must call this routine. Be sure to call this routine even if the request fails. Your component should pass the reference constant that the client program provided with the `refCon` parameter.

The client program must provide a completion routine for all asynchronous write requests. For synchronous requests, client programs should set this parameter to `nil`.

`refCon` Contains a reference constant that your data handler component should provide to the data-handler completion function specified with the `completion` parameter.

For synchronous operations, client programs should set this parameter to 0.

DESCRIPTION

The `DataHWrite` function provides both a synchronous and an asynchronous write interface. Synchronous write operations work like the `DataHPutData` function—the data handler component returns control to the client program only after it has serviced the write request. Asynchronous write operations allow client programs to queue write requests. Your data handler queues the request and immediately returns control to the calling program. After your component actually writes the data, it calls the client program's data-handler completion function.

ERROR CODES

`dataNotOpenForWrite`	–2043	Data reference not open for write

SEE ALSO

Client programs can force your component to write any cached data by calling your component's `DataHFlushData` function.

DataHSetFileSize

Your component sets the size, in bytes, of the current data reference.

```
pascal ComponentResult DataHSetFileSize (DataHandler dh,
                                long fileSize);
```

`dh`	Identifies the calling program's connection to your data handler component.
`fileSize`	Specifies the new size of the container corresponding to the current data reference, in bytes.

DESCRIPTION

The `DataHSetFileSize` function is functionally equivalent to the File Manager's `SetEOF` function. If the client program specifies a new size that is greater than the current size, your component should extend the container to accommodate that new size. If the client program specifies a container size of 0, your component should free all of the space occupied by the container.

DataHGetFileSize

Your component returns the size, in bytes, of the current data reference.

```
pascal ComponentResult DataHGetFileSize (DataHandler dh,
                                long *fileSize);
```

`dh`	Identifies the calling program's connection to your data handler component.
`fileSize`	Contains a pointer to a long. Your component returns the size of the container corresponding to the current data reference, in bytes.

DESCRIPTION

The `DataHGetFileSize` function is functionally equivalent to the File Manager's `GetEOF` function.

DataHCreateFile

Your component creates a new container that meets the specifications of the current data reference.

```
pascal ComponentResult DataHCreateFile (DataHandler dh,
                                        OSType creator,
                                        Boolean deleteExisting);
```

`dh`	Identifies the calling program's connection to your data handler component.
`creator`	Specifies the creator type of the new container. If the client program sets this parameter to 0, your component should choose a reasonable value (for example, `'TVOD'`, the creator type for Apple's movie player).
`deleteExisting`	Indicates whether to delete any existing data. If this parameter is set to `true` and a container already exists for the current data reference, your component should delete that data before creating the new container. If this parameter is set to `false`, your component should preserve any data that resides in the container defined by the current data reference (if there is any).

DataHGetPreferredBlockSize

The `DataHGetPreferredBlockSize` function allows your component to report the block size that it prefers to use when accessing the current data reference.

```
pascal ComponentResult DataHGetPreferredBlockSize
                                        (DataHandler dh,
                                        long *blockSize);
```

`dh`	Identifies the calling program's connection to your data handler component.
`blockSize`	Contains a pointer to a long. Your component returns the size of blocks (in bytes) it prefers to use when accessing the current data reference.

DESCRIPTION

Different devices use different file system block sizes. This function allows your component to report its preferred block size to the client program. Note that the client program is not required to use this block size when making requests. Some clients may, however, try to accommodate your component's preference.

DataHGetFreeSpace

Your component reports the number of bytes available on the device that contains the current data reference.

```
pascal ComponentResult DataHGetFreeSpace (DataHandler dh,
                                unsigned long *freeSize);
```

`dh`	Identifies the calling program's connection to your data handler component.
`freeSize`	Contains a pointer to an unsigned long. Your component returns the number of bytes of free space available on the device that contains the container referred to by the current data reference.

DataHPreextend

Your component allocates new space for the current data reference, enlarging the container.

```
pascal ComponentResult DataHPreextend (DataHandler dh,
                                long maxToAdd,
                                long *spaceAdded);
```

`dh`	Identifies the calling program's connection to your data handler component.
`maxToAdd`	Specifies the amount of space to add to the current data reference, in bytes. If the client program sets this parameter to 0, your component should add as much space as it can.

`spaceAdded`	Contains a pointer to a long. Your component returns the number of bytes it was able to add to the data reference, in bytes.

DESCRIPTION

This function is essentially analogous to the File Manager's `PBAllocContig` function. Your component should allocate contiguous free space. If there is not sufficient contiguous free space to satisfy the request, your component should return a `dskFulErr` error code.

Client programs use this function in order to avoid incurring any space-allocation delay when capturing movie data.

Managing Data Handler Components

Your data handler component provides a number of functions that applications can use to manage their connections to your handler. The most important among these is `DataHTask`, which provides processor time to your handler. Applications should call this function often so that your handler has enough time to do its work.

Applications may call your handler's `DataHPlaybackHints` function in order to provide you with some guidelines about how those applications play to use the current data reference.

The `DataHFlushData` and `DataHFlushCache` functions allow applications to influence how your component manages its stored data.

DataHTask

Client programs call your component's `DataHTask` function in order to cede processor time to your data handler.

```
pascal ComponentResult DataHTask (DataHandler dh);
```

`dh`	Identifies the calling program's connection to your data handler component.

DESCRIPTION

This function is essentially analogous to the Movie Toolbox's `MoviesTask` function. Client programs call this function in order to give your data handler component time to do its work. Your data handler uses this time to do its work. Because client programs will call this function frequently, and especially so during movie playback or capture, your data handler should return control quickly to the client program.

DataHFlushCache

Your component discards the contents of any cached read buffers.

```
pascal ComponentResult DataHFlushCache (DataHandler dh);
```

`dh`	Identifies the calling program's connection to your data handler component.

DESCRIPTION

Client programs may call this function if they have, in some way, changed the container associated with the current data reference on their own. Under these circumstances, data your component may have read and cached in anticipation of future read requests from the client may be invalid.

Note that this function does not invalidate any queued read requests (made by calling your component's `DataHScheduleData` function).

DataHFlushData

Your component forces any data in its write buffers to be written to the device that contains the current data reference.

```
pascal ComponentResult DataHFlushData (DataHandler dh);
```

`dh`	Identifies the calling program's connection to your data handler component.

DESCRIPTION

This function is essentially analogous to the File Manager's `PBFlushFile` function. The client program may call this function after any write operation (either `DataHPutData` or `DataHWrite`). Your component should do what is necessary to make sure that the data is written to the storage device that contains the current data reference.

DataHPlaybackHints

The `DataHPlaybackHints` function allows the client program to provide additional information to your component that you may use to optimize the operation of your data handler.

```
pascal ComponentResult DataHPlaybackHints (DataHandler dh,
                                           long flags,
                                           unsigned long minFileOffset,
```

```
                                    unsigned long maxFileOffset,
                                    long bytesPerSecond);
```

`dh`	Identifies the calling program's connection to your data handler component.
`flags`	Reserved for use by Apple Computer, Inc. Client programs should always set this parameter to 0.
`minFileOffset`	Together with the `maxFileOffset` parameter, specifies the range of data the client program anticipates using from the current data reference. This parameter specifies the earliest byte the program expects to use (that is, the minimum container offset value). If the client expects to access bytes from the beginning of the container, it should set this parameter to 0.
`maxFileOffset`	Specifies the latest byte the program expects to use (that is, the maximum container offset value). If the client expects to use bytes throughout the container, the client should set this parameter to –1.
`bytesPerSecond`	Indicates the rate at which your data handler must read data from the data reference in order to keep up with the client program's anticipated needs.

DESCRIPTION

Your component should be prepared to have this function called more than once for a given data reference. For example, the Movie Toolbox calls this function whenever a movie's playback rate changes. This is a handy way for your data handler to track playback rate changes.

Completion Function

When client programs schedule asynchronous read or write operations (by calling your component's `DataHScheduleData` or `DataHWrite` functions), they furnish your component a data-handler completion function. Your component must call this function when it completes the read or write operation, whether the operation was a success or a failure.

Data-handler Completion Function

The client program's completion function must present the following interface:

```
pascal void DHCompleteProc (Ptr request, long refcon,
                                    OSErr err);
```

`request`	Specifies a pointer to the data that was associated with the read (`DataHScheduleData`) or write (`DataHWrite`) request. The client program uses this pointer to determine which request has completed.
`refcon`	Contains a reference constant that the client program supplied to your data handler component when it made the original request.
`err`	Indicates the success or failure of the operation. If the operation succeeded, set this parameter to 0. Otherwise, specify an appropriate error code.

CHAPTER 10 QUICKTIME ' MUSIC ARCHITECTURE

This chapter describes the QuickTime Music Architecture. This chapter includes descriptions of the data structures and functions that allow your application to control the time-based music features included with QuickTime 2.0. With the QuickTime Music Architecture your application allows users to play, edit, cut, copy and paste movie music data in the same way they work with text and graphic elements today.

QUICKTIME MUSIC ARCHITECTURE OVERVIEW

The QuickTime Music Architecture (QMA) is a set of music events and three layers of software components. Each component layer provides a set of functions that allow your applications or QuickTime Movie music tracks to create and control music elements on the Macintosh. Additional control for external MIDI devices is also provided.

Music "events" are used to specify the Instruments and notes of a musical composition, called a sequence. A group of events is called a "sequence." A sequence of events may define a range of Instruments and their characteristics, along with a sequence of notes and rests which, when interpreted, product the musical composition. Such event sequences can be contained within a QuickTime music track or be produced by your application. QMA interprets and plays the music from the sequence data.

The three layers of QMA provide different levels of access to the actual devices used to create sound. The top-most component layer provides timing for the sequence and minimizes the need to understand and manage the specific details for each synthesizer device. The next component layer plays individual notes to a specified synthesizer device. The lowest component level provides access to synthesizer device specifics.

The available QMA components are the:

- Tune Player
- Note Allocator
- Music Component

The Tune Player component plays a time-ordered sequence of events. The Tune Player negotiates with the Note Allocator, described below, to determine which Music Component to use. For example, if the music score requires a piano, the sequence will request a "piano" resource. QMA provides an Instrument that best "fits" the request. At the top-most layer the sequence is not required to know about the specific Instrument type. The sequence only needs to know that it needs a piano. At the lowest layer however, the Music Component provides specific details about each available sound producing device.

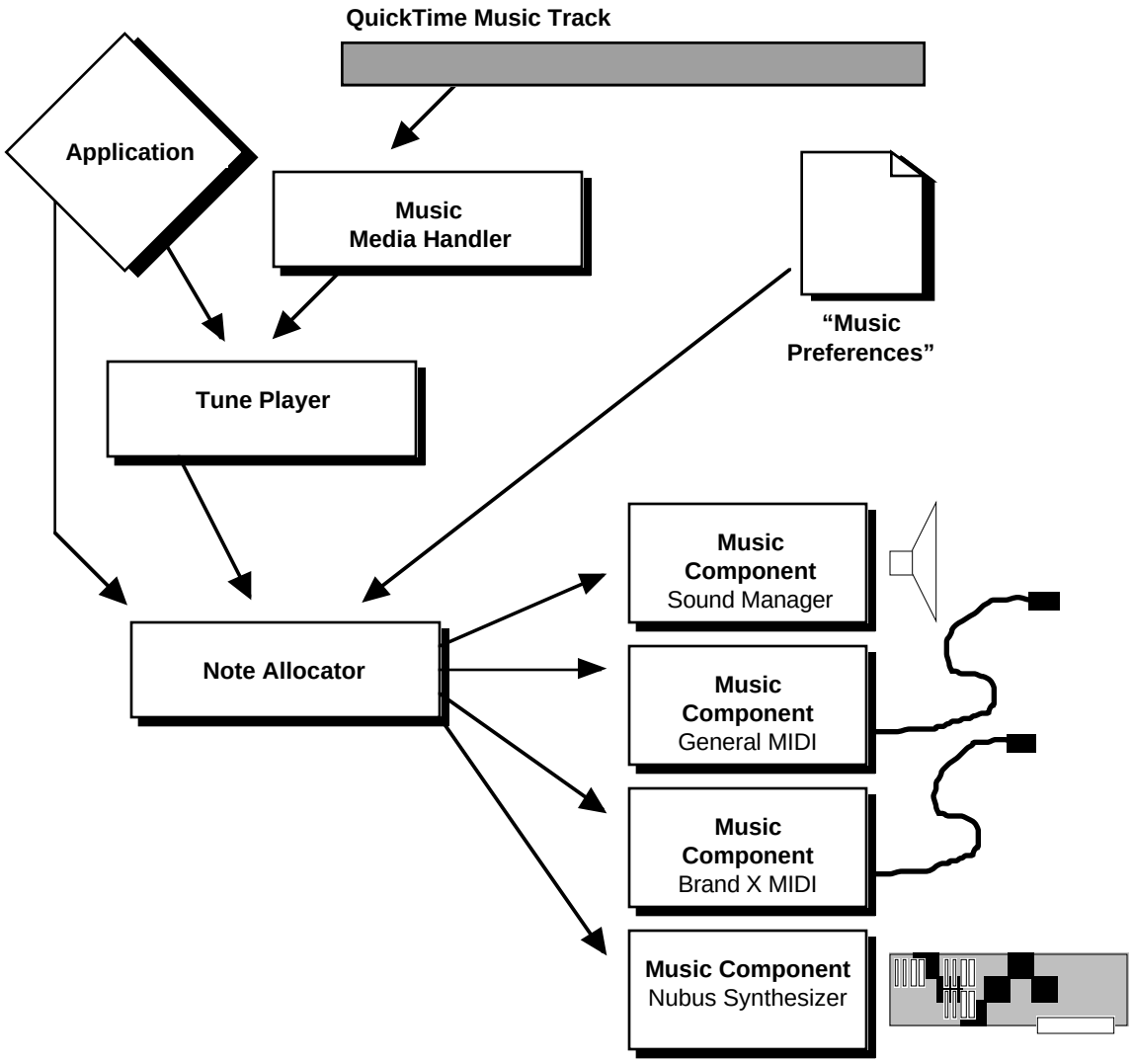
The sequence of events is sent to the Note Allocator which in turn sends them to an appropriate Music Component. The Tune Player handles all aspects of timing as defined by the sequence of events.

The Note Allocator component can be used to play individual notes from a synthesizer. Unlike the Tune Player, there are no timing services. The Note Allocator also contains miscellaneous functions to handle external MIDI devices, create and maintain a database of Music Components, and provide special functions to gain access to the details of each Music Component.

Music Components are sound-playing software or software components utilizing external hardware devices to produce music. These components either produce the sounds through software-only means or interact with hardware devices which produce sound.

As the following diagram illustrates, QMA can be accessed by QuickTime Music tracks or by applications. QuickTime Music tracks can contain a sequence of events and use a standard Music Media Handler to access the Tune Player.

An application will usually access QMA through the Tune Player or the Note Allocator. Some applications will access the Music Components directly but this is usually unnecessary.



GENERAL TERMINOLOGY

DURATION:

Music sequences contain notes and rests. The length in time of either is described by its duration. A duration is defined by a number of units-per-second.

INSTRUMENT:

A particular sound on a synthesizer (see synthesizer below).

KNOB:

A user-modifiable Instrument adjustment value.

MICROTONES:

The musical scale most often used in modern Western music is the 12 tone "equal tempered" scale. This scale divides each octave into 12 available pitches (frequencies), called "semitones." Any pitch that lies between two semitones is called a "microtone." The QuickTime Music Architecture lets you specify 255 microtones between each pair of semitones.

MIDI:

Musical Instrument Digital Interface is a standard serial protocol for communication between electronic musical devices.

MUSIC COMPONENT:

A software component, which adheres to the QMA Music Component interface standard, to produce sound either through the Macintosh's built-in speaker or by controlling an external hardware device.

MUSIC TRACK:

A sequence of QMA events used to describe music in terms of the notes, rests and Instruments to be used.

NOTE:

A sound defined by its pitch, volume (velocity) and duration.

NOTE CHANNEL:

An abstract reference to a synthesizer Part which can play notes.

POLYPHONY:

A number of simultaneous musical notes. The polyphony of a synthesizer is the maximum number of notes it can play at one point in time. The polyphony of a music track is the maximum number of notes it ever plays at one point in time.

PART:

A single assignable Instrument slot within a synthesizer. A synthesizer contains a number of Parts. This maximum number defines the synthesizer's timbrality. Each Part can be set to one Instrument. An initialized Part can be modified through its Knobs to produce a unique Instrument. Modified Parts may be saved as new Instruments and later recalled.

PITCH:

The relative position of a note in a scale as determined by its frequency. Any of various standards that establish a frequency for each musical note, used in the tuning of Instruments.

SYNTHESIZER:

A software or hardware device capable of creating sounds. To be used by QuickTime, a synthesizer must have a corresponding Music Component which provides the software interface to that synthesizer.

TIMBRE:

The quality of a sound which makes it uniquely identifiable regardless of the sound's pitch or volume. The unique qualities or attributes that make the sounds of a piano, tuba or oboe, all playing the same note, uniquely identifiable.

TUNE PLAYER:

A software component used to assign available Instruments and to play a sequence of QMA music events. The Tune Player provides abstract access to Instruments and system timing for long sequences of music.

TRACK:

In the context of QMA, a track is a sequence of music events contained in a QuickTime movie track.

VOICE:

Voices and oscillators are interchangeable terms. The maximum number of voices available to a synthesizer defines its polyphony.

VOLUME:

The amplitude or loudness of a sound. The audio level, described by QMA as a number from 0.0 to 1.0, used to adjust the output of either a Part, an entire synthesizer or both.

ADVANTAGES OF QUICKTIME MUSIC ARCHITECTURE VS. MIDI

- QMA is not limited to MIDI's 16 simultaneous timbres
- It supports generalized access to synthesizer specific features.
- Devices can report their details about device specifics.
- QMA offers a natural implementation of microtonal scales.
- QMA file format is simpler than standard MIDI format.

The QMA's API has no limitation on the number of timbres (Parts) available to an application or music track. MIDI limits the number of timbres to 16.

The QMA supports generalized access to synthesizers. This ability eliminates the requirements for an application to support a range of specific devices. In some case, however, an application may need greater control to a specific type of synthesizer. Access to a particular Music Components provides this type of control.

In addition to producing standard equal tempered notes, the QMA file format allows 256 microtonal values between each standard note.

COMPONENTS OF QUICKTIME MUSIC ARCHITECTURE

The three layers of the QuickTime Music Architecture are the:

- Tune Player
- Note Allocator
- Music Component

The Tune Player component of the QuickTime Music Architecture can be used to play sequences of notes and Rest event data in a simple manner.

The Note Allocator provides services to play individual notes out of a synthesizer. Unlike the Tune Player, the Note Allocator has no ability to provide sequence timing.

The Music Component is a software component which is used to produce sound through software-only algorithms or through an interface to external hardware. Generally an application will not need to call the Music Component directly. Usually calls through the Note Allocator will provide adequate service. Macintosh provides the Software Synthesizer and General MIDI Music Components.

Tune Player

The Tune Player plays sequences of music. It also allocates the necessary note channels for a particular sequence.

In addition, the Tune Player provides the timing support necessary to interpret and play a music sequence, compared to the Note Allocator which has no timing support.

Any number of sequences may be played simultaneously as long as there is sufficient polyphony (voices) within the specific Music Component allocated by the Tune Player.

Sequences can be played from beginning to end or only a portion of a sequence can be played. An additional sequence, or sequence section, may be queued-up while one is currently being played. Queuing sequences can provide a seamless way to transition between sections.

The Tune Player is implemented as a component. Each instance of the Tune Player component can play a sequence.

Note Allocator

The Note Allocator provides a way to access and manage the available synthesizers without the need to understand a synthesizer’s specific details.

The Note Allocator, unlike the Tune Player, provides no timing related features to manage the sequence. The Note Allocator's features are similar to the Music Component's, as described below, although more generalized.

The Note Allocator's services fall into three categories:

- Note channel allocation and use
- System configuration
- Miscellaneous interface tools

Note channel allocation will create a note channel by selecting and allocating a Part, within a synthesizer, based on the tone requested, provides detailed information about an allocated note channel, and allows configuration of, and access to, external MIDI devices.

In addition, note channel allocation provides features to reserve in advance, and release when finished or temporarily not needed, resources required to play a sequence. A sequence's overall volume can be adjusted and the note channel can be engaged (default) or disengaged while playing.

Note channel use will play individual notes, apply a specified controller to the allocated note channel, provide access to Knobs to adjust a Part's characteristics, select an Instrument based on a required tone, and modify or change the Instrument type on an existing note channel.

System configuration provides services which create and maintain a database of Music Components, save configuration information in a “Music Preferences” file and establish connections to external MIDI devices.

The miscellaneous interface tools provide a set of user interface dialogs to select individual Instruments, select Instruments within an arrangement and to provide copyright information for a particular Instrument.

Music Component

A Music Component is a software component used to produce sound. The method a component uses to produce sound depends on the component.

An application generally accesses a Music Component through the Note Allocator or Tune Player. Applications do not usually call music components directly.

The standard Macintosh Music Components include a software synthesizer and a General MIDI music component.

In addition to these standard components, a Music Component can be created to control an external hardware synthesizer. In this case, the sound is produced by the external synthesizer, not the software component.

A Music Component provides synthesizer access similar to that of the Note Channel. The Note Channel's access, however, is generalized and indirect. The Music Component can directly access a particular synthesizer's features and controls. This type of access is available only through a Music Component.

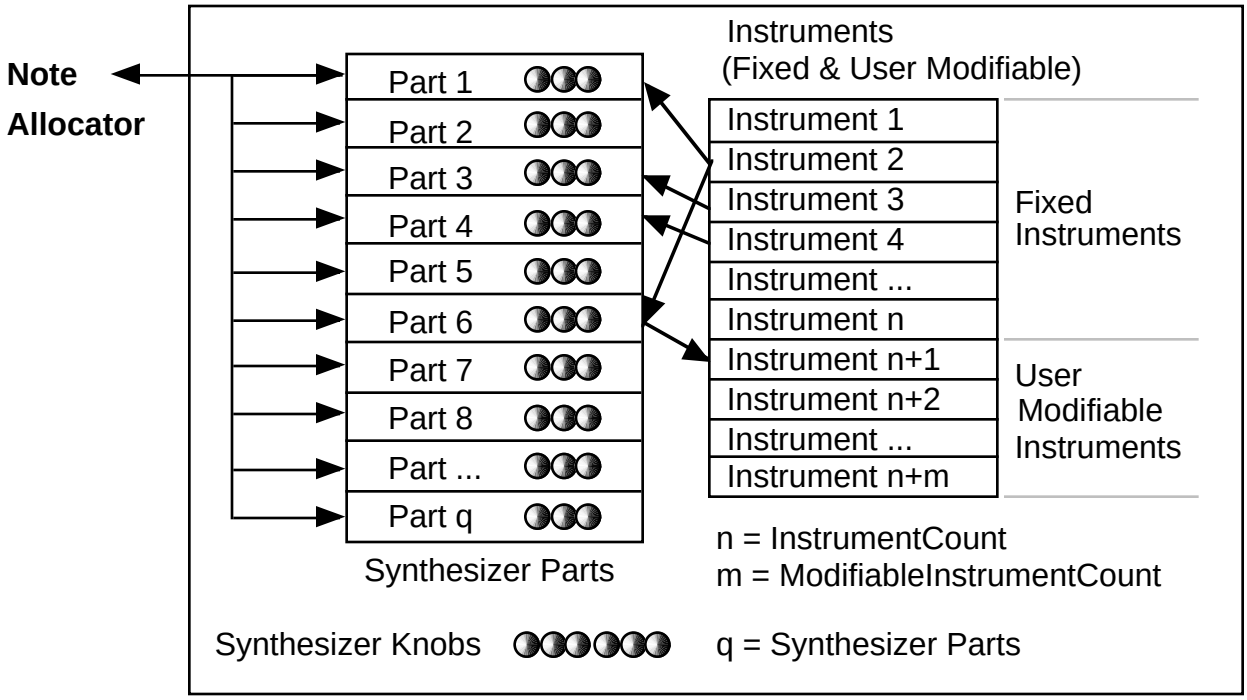
To better understand the role of a Music Component it's important to understand the features of a generic QMA synthesizer.

A synthesizer contains a number of Parts and Instruments. An Instrument is a very specific description of the type of sound produced. Parts can be thought of as slots in which the user installs particular Instruments.

Instruments are grouped into fixed (built-in) and user-modifiable Instruments*See* Modifiable Instruments. Instruments installed or loaded into a Part can be used as-is or modified and saved into a user-modifiable Instrument for later recall.

An Instrument is accessible only after it is loaded into one of the synthesizer's Parts. An Instrument loaded into a Part can be modified by changing the value of one of its Knobs, and saving to one of the Modifiable Instruments using a new Instrument name. Instruments cannot be saved to a Fixed Instrument location.

The diagram below illustrates the internal model of a Music Component (described here as a generic synthesizer). The illustration shows the total number of Parts available from the synthesizer, a group of fixed Instruments, 1 through n, and a group of user modifiable Instruments, n+1 through n+m.



Generic Synthesizer Model

The illustration above shows the synthesizer's Part 1 loaded with the Fixed Instrument 2. Once an Instrument is loaded into a Part it may be modified and subsequently saved, with a new Instrument name, to a user modifiable Instrument.

An example of a user modifiable Instrument is shown using Part 6. It uses the same Instrument as Part 1. One Instrument can be used by two separate Parts. After an Instrument is loaded into multiple Parts, either Part can be modified, through its Knobs, to produce a unique variation from the original Instrument. In this example, Part 6 is saved to the user modifiable Instrument n+1. Parts cannot be saved to the fixed Instrument bank.

Each Part has its own set of Knobs. There is another set of Knobs that apply to the entire synthesizer and not to a particular Instrument. These Knobs are typically for controlling effects like audio effects (such as reverb) that may be built into a device. In addition to the synthesizer Knobs are controllers which will also modify the characteristics of the synthesizer.

The Music Component services fall into 4 categories:

- Synthesizer access
- Instrument access
- Part access
- Synthesizer timing

The Music Component synthesizer access functions provide services to obtain specific information about the current synthesizer and obtain an Instrument which best fits a requested type of sound. The synthesizer access will also play a note with a specified pitch, volume and duration, get and set a particular synthesizer's Knob, obtain default Knob information, and get and set external MIDI procedure name entry points.

The Music Component Instrument access provides services to initialize a Part to a specified Instrument, to create and return an organized group of available synthesizer Instrument and Drum names, and return the Instrument number assigned to the specified Part. The Instrument access also stores modified Instruments from a Part into the modifiable Instrument store, gets detailed information about each Instrument available from the synthesizer, and returns detailed default Instrument Knob settings.

The Music Component Part access provides services to get and set synthesizer Part parameters, to get and set a Part's human interface name, to get and set the value for a particular Part Knob, to reset the Part to a default state and get and apply controller values to individual Parts modifying their characteristics.

The Music Component synthesizer timing provides services to get and modify the master reference timer used by the synthesizer.

EVENT SEQUENCE FORMAT

QMA defines music as a sequence of events. The events described in this section initialize and modify sound producing music devices and define the notes and rests to be played.

A sequence of events is required to produce music. The sequence of events is generally contained within either a QuickTime movie track (which uses a media handler to provide access to the Tune Player), or an application containing a sequence of events. The application will pass them directly to the Tune Player.

> ***Note:*** Using the MoviePlayer a standard MIDI sequence file will automatically be converted to a QuickTime music track sequence. Refer the “Conversion of Standard MIDI” chapter for additional details.

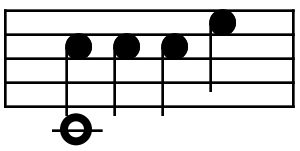
Events are constructed as a group of long words. The upper 1st four bits (nibble) of an event's long word defines its type.

1st Nibble	Long Words	Event Type
000x	1	Rest
001x	1	Note
010x	1	Controller
011x	1	Marker
1000	2	(reserved)
1001	2	Note
1010	2	Controller
1011	2	Knobs
1100	2	(reserved)
1101	2	(reserved)
1110	2	(reserved)
1111	n	General

It's important to understand that the Rest event specifies the duration before interpreting the next event in the stream. A Rest event does not specify an independent period of silence. The Rest event defines when to act on the next event in sequence. A Note event will define its own end by the specific duration contained within the Note event.

Both the Note durations and the Rest durations are specified in units of the Tune Player's time scale (default of 1/600ths of a second).

Consider the following musical fragment.



Assuming 120 beats-per-minute, and a Tune Player's scale of 600, each quarter note's duration is 300. The music track data to describe this fragment would appear as follows.

NOTE Part 0, pitch 60, duration 1200 plays for four beats

NOTE Part 0, pitch 72, duration 300 plays for one beat

REST duration 300 delays start of next note

NOTE Part 0, pitch 72, duration 300 plays for one beat

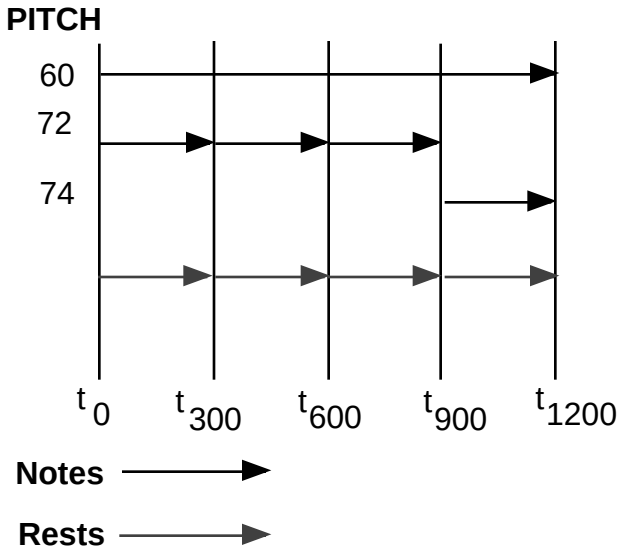
REST duration 300 delays start of next note

NOTE Part 0, pitch 72, duration 300 plays for one beat

REST duration 300 delays start of next note

NOTE Part 0, pitch 74, duration 300 plays for one beat

REST duration 300 delays start of next note



The General event is used to specify the types of Instruments or sounds used for the subsequent Note events.

The Note event causes a specific Instrument, previously defined by a General event, to play a note at a particular pitch and velocity for a specified duration of time.

Additional event types allow sequences to apply controller effects to Instruments, define rests and modify Instrument Knob values. The entire sequence is closed with the End Marker event. The End Marker event is currently limited to this "end of sequence" identifier. Future functionality is intended and reserved.

In most cases, the standard note and Controller events (2 long words) will provide sufficient functionality for an application's requirements.

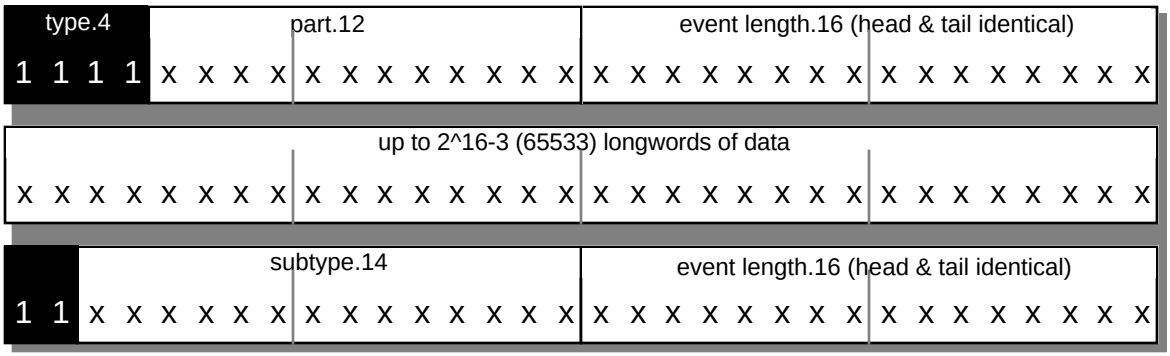
The Extended Note event provides greater pitch range and microtonal note control for music that requires these capabilities.

The Extended Controller event expands the number of Instruments and controller values an application can specify.

General Event

The General event is currently only used to inform QMA of a synthesizer to be use by subsequent events. A subtype of 1 must be used. The Tune Player call, `TuneSetHeader()`, receives the General event described below.

General Event (Variable Length)



General event type field	1st nibble value = 1111
Part	Instrument index number
Event length	head: number of words in event
Variable data words	noteRequest structure below
Subtype	noteRequest subtype must be 1
Event length	tail: must be identical to head
Event tail	1st nibble of last word = 11XX

```
typedef struct {
   short polyphony;  /* Preferred number of voices */
   ToneDescription tone;
} NoteRequest;

struct ToneDescription {
   OSType   synthesizerType;
   Str31    synthesizerName;
   Str31    instrumentName;
   long     instrumentNumber;
   long     gmNumber;
};
```

The Part number bit field is uniquely defined and set by the application. The unique Part number is used in all subsequent events where the Part is referenced. For example, to play a note the application will use this Part number to specify which Instrument will play the note. The General event allows specifying Part numbers of up to 12 bits. The standard note and controller events allow Part number of up to 5 bits in length.

The event length bit fields contained in the first and last words of the message are identical and are used as a message format check and to move both forward and backward through the message.

The variable length data field contains information unique to the type of General event. There is currently only a note request General event. The note request structure used to define the Instrument or Part and is contained within the variable length data field.

The subtype bit field indicates the type of General event. Currently there is only a note request General event with a subtype of 1. If the subtype is any other value, the event is ignored.

Macro calls are used to stuff the General event's head and tail long words, but not the structures described above:

```
_StuffGeneralEvent(w1, w2, instrument, subType, length)
```

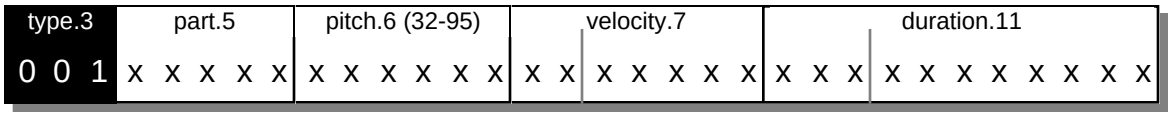
Macros are used to extract field values from the event's head and tail long words.

```
_XInstrument(m, l)
_GeneralSubtype(m, l)
_GeneralLength(m, l)
```

Note Event

The standard Note event (as compared with the Extended Note event) supports most music requirements. The Note event allows up to 32 Instruments and supports the traditional equal tempered scale.

Note



Note Event type field	1st nibble value = 001X
Part	Part index number
Pitch	numeric value of 0-63, mapped as 32-95
Velocity	0-127, 0 = no audible response
Duration	Units of time the note will occur

The Part field is the Instrument number initially used during the `TuneSetHeader()` call.

The pitch bit field allows a range from 0-63 which is mapped to the values 32-95 representing the traditional equal tempered scale. For example, the value 23 (mapped to 60) is middle C.

The velocity bit field allows a range from 0-127 and translates into the volume of the specified Part. A velocity value of 0 produces silences.

The duration bit field defines the number of units of time during which the Part will play the pitch. The units of time are defined by the media time scale or Tune Player time scale.

Macro call used to stuff the Note event's long word:

```
_StuffNoteEvent(x, instrument, pitch, volume, duration)
```

Macro calls used to extract fields from the Note event's long word:

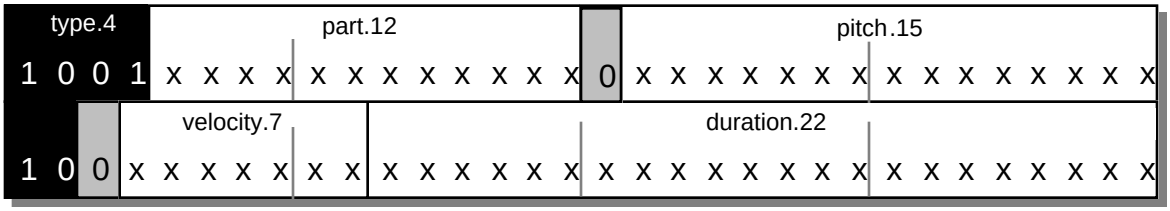
```
_Instrument(x)
_NotePitch(x)
_NoteVelocity(x)
_NoteVolume(x)
_NoteDuration(x)
```

> ***Note:*** The standard Note event does not allow microtonal values, pitches below 32 or above 95. For these extended features use the Extended Note event.

Extended Note Event

The Extended Note event, compared to the standard Note event, provides a wider range of pitch values (traditional equal tempered scale), microtonal values to define any pitch, and extended note duration. The Extended Note event requires two long words; the standard Note event requires only one.

Extended Note



Extended Note type field	1st nibble value = 1001
Instrument	extended Instruments index

Pitch	0-127 standard pitch , 60 = middle C 0x01.00 .. 0x7f.00 allowing 256 microtonal divisions between each notes in the traditional equal tempered scale.
Note Duration	extended note duration
Note Velocity	0-127 where 0 = silence
Event tail	1st nibble of last word = 10XX

The Part number bit field is the Instrument index assigned to the Part when initialized by the General event call to `TuneSetHeader`.

If the pitch field is less than 128, then it is interpreted as an integer pitch where 60 is middle C. If the pitch is 128 or greater, it is treated as a fixed pitch.

Microtonal pitch values are produced when the 15 bits of the pitch field are split into an upper 7 bits to define the pitch and a lower 8 bits to define the pitch's fractional portion. This is represented by 0x01.00 to 0x7F.00: where 0x01-0x7F defines the standard equal tempered note with the lower 8 bits defining 256 microtonal divisions between the standard notes.

Macro call used to stuff the extended Note event's long words:

```
_StuffXNoteEvent(w1, w2, instrument, pitch, volume, duration)
```

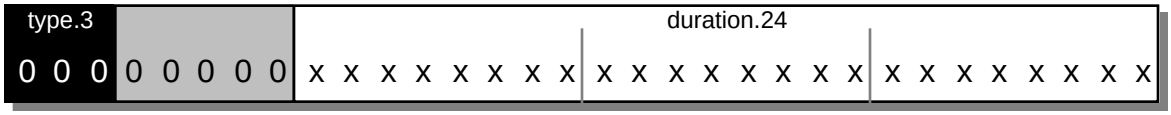
Macro calls used to extract fields from the extended Note event's long words:

```
_XInstrument(m, l)
_XNotePitch(m, l)
_XNoteVelocity(m, 1)
_XNoteVolume(m, l)
_XNoteDuration(m, l)
```

Rest Event

The Rest event specifies the period of time, defined by either the media time scale or the Tune Player time scale, until the next Note event in the sequence will be played.

Rest



Rest Event type field	1st nibble value = 000X
Duration	Duration in units defined by media time scale or Tune Player time scale.

The duration bit field specifies the number of units of time until the next Note event is played.

Macro call used to stuff the Rest event's long word:

```
_StuffRestEvent(x, duration)
```

Macro call used to extract the Rest event's duration value:

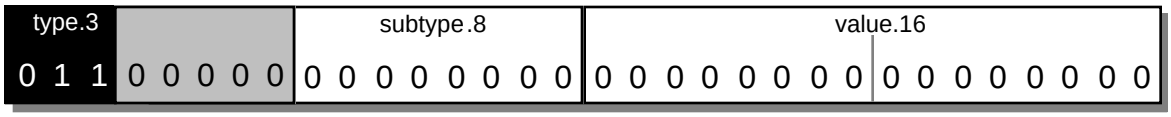
```
_RestDuration(x)
```

> ***Note:*** It is important to understand that the Rest events are not used to cause silence in a sequence but to define the start of subsequent Note events.

End Marker Event

The End Marker event has subtype and value fields containing zero.

End Marker



End Marker event type field	1st nibble value = 011X
End Marker subtype	8 bit unsigned subtype = 0
End Marker value	16 bit signed value = 0

The End Marker subtype bit field must contain zeros.

The End Marker value bit field must contain zeros.

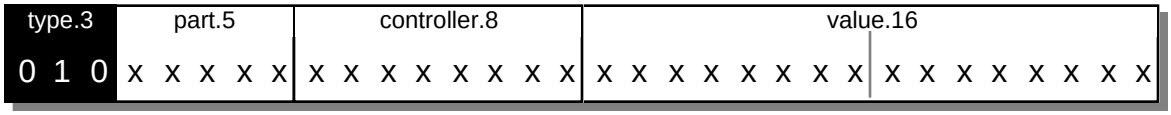
Macro call used to extract fields from the Rest event's long word:

```
_MarkerSubtype(x)
_MarkerValue(x)
```

Controller Event

The Controller event changes the value of a controller on a specified Part.

Controller



Controller event type field	1st nibble value = 010X
Part	Instrument index number
Controller	controller to be applied to Instrument
Value	8.8 bit fixed point signed controller specific value

Currently defined controller types:

```
kControllerModulationWheel    0-7F.FF max effect

kControllerVolume             0-7f.ff max effect (default)

kControllerPan                0=left, 1.00=right

kControllerPitchBend          0x0100 raises the pitch by one
                                 semi-tone and 0xFF00 lowers by one
                                 semi-tone.
```

The Part field is the Instrument number initially used during the `TuneSetHeader()` call.

The controller bit field is a value which describes the type of controller used by the Part.

The value bit field is specific to the selected controller.

Macro call used to stuff the controller event's long word:

```
_StuffControlEvent(x, instrument, control, value)
```

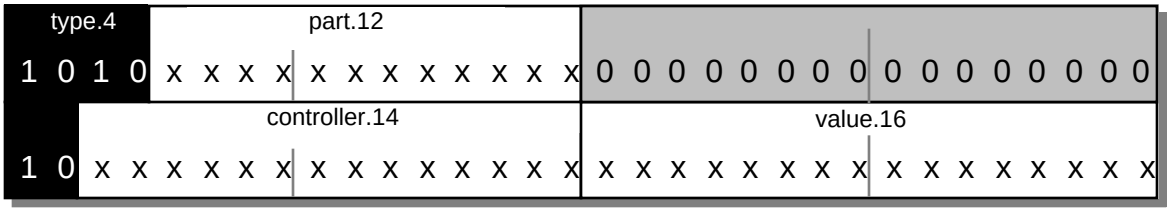
Macro calls used to extract fields from the controller event's long word:

```
_Instrument(x)
_ControlController(x)
_ControlValue(x)
```

Extended Controller Event

The Controller event changes the value of a controller on a specified Part. The Extended Controller event allows Parts and controllers beyond the range of the standard Controller event.

Extended Controller



Extended Controller type field	1st nibble value = 1010
Part	Instrument index for controller
Controller	Controller for Instrument
Value	Signed controller specific value
Event tail	1st nibble of last word = 10XX

The Part field is the Instrument number initially used during the `TuneSetHeader()` call.

The controller bit field is a value which describes the type of controller to be used by the Part.

The value bit field is specific to the selected controller.

Macro call used to stuff the Extended Controller event's long words:

```
_StuffXControlEvent(w1, w2, instrument, control, value)
```

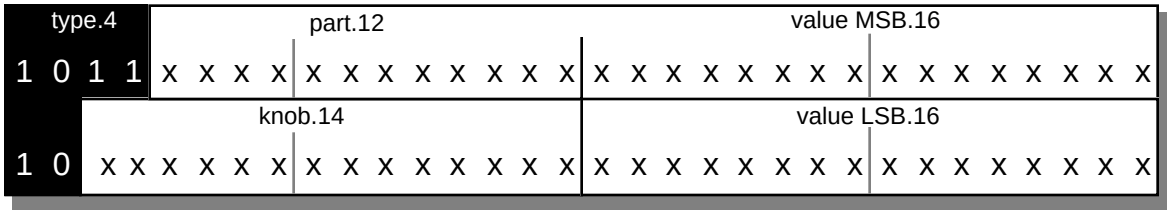
Macro calls used to extract fields from the Extended Controller event's long words:

```
_XInstrument(m, l)
_XControlController(m, l)
_XControlValue(m, l)
```

Knob Event

The Knob event is used to modify a particular parameter within a specified Part.

Knob



Knob event type field	1st nibble value = 1011
Part	Instrument index number
Knob number	Knob number within specified Part
Knob value (LSW (0-15))	lower 16 bits of Knob value
Knob value (MSW (16-31))	upper 16 bits of Knob value
Event tail	1st nibble of last word = 10XX

The Part field is the Instrument number initially used during the `TuneSetHeader()` call.

The 32 bit value composed of the lower 16 and upper 16 bit field values is used to alter the specified Knob.

The Knob bit field specifies which Knob is effected by the value.

Macro call used to stuff the Knob event's long words:

```
_StuffKnobEvent(w1, w2, instrument, knob, value)
```

Macro calls used to extract fields from the Knob event's long words:

```
_XInstrument(m, l)
_KnobValue(m, l)
_KnobKnob(m, l)
```

COMPONENT INTERFACES

The Note Allocator, Tune Player and Music Component APIs are described in the following sections.

Tune Player

The QuickTime Music Architecture Tune Player component is used to play sequences of notes and Rest event data in a straightforward manner.

An application need only open an instance of the Tune Player component, call `TuneSetHeader()` with the appropriate header data, and call `TuneQueue()` with the desired sequence data.

The Tune Player will handle all timing necessary to play a sequence of notes and rests. In addition, the Tune Player provides services to set the volume, and to stop and restart an active sequence.

> ***Note:*** It is often easier to use the QuickTime Toolbox to play a movie that contains music data rather than utilizing the Tune Player directly.

The Tune Player component provides a layer of abstraction from the actual underlying synthesizer components. This allows the application to select musical components at a general level and allows the Tune Player to pick the Instrument that is both available and best fits the application's request.

The Tune Player is also used to specify details about a tune sequence, modify time base and time scale and get detailed information about actual Instrument selections.

Sequence Data

TuneSetHeader

The `TuneSetHeader ( )` function prepares the Tune Player to accept subsequent music sequences by defining one or more Parts used by sequence Note events.

```
pascal ComponentResult TuneSetHeader
                                (TunePlayer tp,
                                unsigned long *header);
```

`tp`	You obtain the Tune Player identifier from the Components Manager's OpenComponent function. See the chapter “Component Manager” in Inside Macintosh: More Macintosh Toolbox for details.
`*header`	A pointer to a list of NoteRequest `(General, subtype1)` events terminated by an end marker.

DESCRIPTION

The `TuneSetHeader ( )` function is the first QMA call in a music sequence. The `*header` parameter points to one or more initialized General events.

The General event, described above, is composed of a group of long words and used to define the Parts available to subsequent Note events by the `TuneQueue()` calls. The `*header` parameter must conclude with an End Marker event.

Only one call to `TuneSetHeader()` is required. Each `TuneSetHeader()` call resets all previous General events.

ERROR CODES

```
noteChannelNotAllocatedErr
tuneParseErr
NoteAllocator errors
```

TuneQueue

The `TuneQueue()` function places a sequence of events into the play queue to be played.

```
pascal ComponentResult TuneQueue
                                (TunePlayer tp,
                                unsigned long *tune,
                                Fixed tuneRate,
                                unsigned long
                                tuneStartPosition,
                                unsigned long
                                tuneStopPosition,
                                unsigned long queueFlags,
                                TuneCallBackUPP callBackProc,
                                long refCon)
```

`tp` You obtain the Tune Player identifier from the Components Manager's OpenComponent function. See the chapter “Component Manager” in Inside Macintosh: More Macintosh Toolbox for details.

`*tune` Pointer to array of events, terminated by an end marker.

`tuneRate` Fixed point speed at which to play the sequence. 0x00010000 is the “normal” speed.

`tuneStartPosition`
Sequence starting time.

`tuneStopPosition`
Sequence ending time.

`queueFlags`

`kTuneStartNow` Start after buffer implied. Play even if another sequence is playing.

`kTuneDontClipNotes`
Allow notes to finish durations outside sample.

`kTuneExcludeEdgeNotes`
Don't play notes that start at end of tune.

`kTuneQuickStart`
Leave all controllers where they are, ignore start time.

`kTuneLoopUntil`
: Loop a queued tune if there's nothing else in the queue.

`callBackProc`
: Points to your callback function.
Your callback function must have the following form:

```
pascal void MyCallBackProc
(QTCallBack cb, long refcon);
```

See “Callback Event Functions” on page 2-364 for details.

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

DESCRIPTION

The `tuneStartPosition` and `tuneStopPosition` specify, in time units numbered from zero for the beginning of the sequence, which part of the queued sequence to play. To play all of it, pass 0 and 0xFFFFFFFF respectively.

If queueFlags = kTuneStartNow, the sequence will immediately begin playing. If there is a sequence currently playing, the newly queued sequence will begin as soon as the active sequence ends.

ERROR CODES

```
tunePlayerFullErr
TimeBase errors
```

Sequence Control

The following functions provide control over the Tune Player's current music sequence.

TuneStop

The `TuneStop` function stops a currently playing sequence.

```
pascal ComponentResult TuneStop
                                (TunePlayer tp,
                                long stopFlags);
```

`tp`	You obtain the Tune Player identifier from the Components Manager's OpenComponent function. See the chapter “Component Manager” in Inside Macintosh: More Macintosh Toolbox for details.
`stopFlags`	Must be zero.

ERROR CODES

None

TuneGetVolume

The `TuneGetVolume` function returns the volume associated with the entire sequence.

```
pascal ComponentResult TuneGetVolume
                                (TunePlayer tp);
```

`tp`	You obtain the Tune Player identifier from the Components Manager's OpenComponent function. See the chapter “Component Manager” in Inside Macintosh: More Macintosh Toolbox for details.

DESCRIPTION

The `TuneGetVolume` function's return value holds a value from 0.0 to 1.0. Individual Instruments within the sequence maintain their current volume levels.

ERROR CODES

None

TuneSetVolume

The `TuneSetVolume` function sets the volume for the entire sequence.

```
pascal ComponentResult TuneSetVolume
                                (TunePlayer tp, Fixed volume);
```

`tp`	You obtain the Tune Player identifier from the Components Manager's OpenComponent function. See the chapter “Component Manager” in Inside Macintosh: More Macintosh Toolbox for details.
`volume`	16.16 Fixed.

DESCRIPTION

The `TuneSetVolume` function sets the volume level of the active sequence to the value of the `volume` parameter ranging from 0.0 to 1.0.

Note: Individual Instruments within the sequence can maintain independent volume levels.

ERROR CODES

`NoteAllocator` errors.

TuneGetTimeBase

The `TuneGetTimeBase` function returns the current sequence TimeBase. (TimeBase calls are described in the QuickTime tool box (need vol name))

```
pascal ComponentResult TuneGetTimeBase
                                (TunePlayer tp, TimeBase *tb);
```

`tp`	You obtain the Tune Player identifier from the Components Manager's OpenComponent function. See the chapter “Component Manager” in Inside Macintosh: More Macintosh Toolbox for details.
`*tb`	An initialized TimeBase object.

DESCRIPTION

The `TuneGetTimeBase` function returns the current TimeBase value used to control the sequence timing. The sequence may be controlled in several ways through its timebase. The rate of playback may be changed, or the TimeBase may be slaved to a different clock or TimeBase than the default of real time.

ERROR CODES

none

TuneGetTimeScale

The `TuneGetTimeScale` function returns the current time scale, in units-per-second, for the specified Tune Player instance.

```
pascal ComponentResult TuneGetTimeScale
                                (TunePlayer tp,
                                TimeScale *scale);
```

`tp`	You obtain the Tune Player identifier from the Components Manager's OpenComponent function. See the chapter “Component Manager” in Inside Macintosh: More Macintosh Toolbox for details.
`*scale`	An initialized TimeScale object.

ERROR CODES

none

TuneSetTimeScale

The `TuneSetTimeScale` function sets the time scale, in units-per-second, used by for the specified Tune Player instance.

```
pascal ComponentResult TuneSetTimeScale
                                (TunePlayer tp,
                                TimeScale scale);
```

`tp`	You obtain the Tune Player identifier from the Components Manager's OpenComponent function. See the chapter “Component Manager” in Inside Macintosh: More Macintosh Toolbox for details.
`scale`	The time scale value to be used.

DESCRIPTION

The `TuneSetTimeScale` function sets the time scale data used by the Tune Player's sequence data when interpreting time based events.

ERROR CODES

none

TuneInstant

The `TuneInstant` function plays the particular sequence events active at the position specified by TunePosition.

```
pascal ComponentResult TuneInstant
                                (TunePlayer tp,
                                unsigned long *tune,
                                long tunePosition)
```

`tp`	You obtain the Tune Player identifier from the Components Manager's OpenComponent function. See the chapter “Component Manager” in Inside Macintosh: More Macintosh Toolbox for details.
`*tune`	Pointer to tune sequence data.
`tunePosition`	Position within tune sequence data.

DESCRIPTION

The `TuneInstant` function plays the notes that are “on” at the specified point in the sequence. The notes are started then left playing upon return. The notes may be silenced by calling TuneStop. This call is useful for enabling user “scrubbing” on a sequence.

ERROR CODES

none

TunePreroll

The `TunePreroll` function attempts to lock down all Tune Player resources necessary in preparation of playing Tune Player sequence data.

```
pascal ComponentResult TunePreroll
                                (TunePlayer tp);
```

`tp`	You obtain the Tune Player identifier from the Components Manager's OpenComponent function. See the chapter “Component Manager” in Inside Macintosh: More Macintosh Toolbox for details.

DESCRIPTION

The `TunePreroll` function attempts to reserve note channels for each Part in the sequence.

ERROR CODES

`NAPreroll` errors

TuneUnroll

The `TuneUnroll` function releases any note channels resources that may have been locked down by previous calls to `TunePreRoll` for this Tune Player.

```
pascal ComponentResult TuneUnroll
                                (TunePlayer tp);
```

`tp`	You obtain the Tune Player identifier from the Components Manager's OpenComponent function. See the chapter “Component Manager” in Inside Macintosh: More Macintosh Toolbox for details.

ERROR CODES

`NAUnroll` errors
Channel Information

TuneGetIndexedNoteChannel

The `TuneGetIndexedNoteChannel` function returns information about the actual Instrument associated with the index passed (refer to `TuneSetHeader`).

```
pascal ComponentResult TuneGetIndexedNoteChannel
                                (TunePlayer tp, short i,
                                NoteChannel *nc);
```

`tp`	You obtain the Tune Player identifier from the Components Manager's OpenComponent function. See the chapter “Component Manager” in Inside Macintosh: More Macintosh Toolbox for details.
`i`	Note channel index.
`*nc`	Allocated initialized note channel.

DESCRIPTION

The Tune Player allocates note channels that best satisfy the requested Instrument in the tune header. The application may use this call to determine which music device was actually used for each note channel.

The index is defined by the application and used initially in the `TuneSetHeader ()` call in the General event. The resulting note channel is used by the `NAnoteChannelInfo ()` call allowing access to the actual music component allocated by the Tune Player.

ERROR CODES

none

TuneGetStatus

The `TuneGetStatus` returns an initialized structure describing the state of the Tune Player instance.

```
pascal ComponentResult TuneGetStatus
                              (TunePlayer tp,
                              TuneStatus *status);
```

`tp`	You obtain the Tune Player identifier from the Components Manager's OpenComponent function. See the chapter “Component Manager” in Inside Macintosh: More Macintosh Toolbox for details.
`*status`	An initialized TuneStatus structure.

```
struct TuneStatus {
   unsigned long *tune;    / sequence starting event
   unsigned long *tunePtr; / event currently playing
   TimeValue time;         / current rel to start
   short queueCount;       / number of seq queued
   short queueSpots;       / number of avail slots
   TimeValue queueTime;    / total time used / queue
   long reserved[3];
};
```

ERROR CODES

none

Note Allocator

To play a single note, an application must open an instance of the Note Allocator component and call `NANewNoteChannel()` with a `NoteRequest` structure - typically to request a standard Instrument within the General MIDI Instrument set (refer to the Appendix). With an open note channel, the application can call `NAPlayNote()` while specifying the note's pitch and velocity. The note will then be played and remain playing until a second call to `NAPlayNote()` is made specifying the same pitch, but with a velocity of zero. The velocity of zero will cause the note to stop.

There are calls for registering and unregistering a Music Component. During registration, the connections for that device are specified (typically, the connections are the MIDI Manager port and client IDs). There is also a call for querying the Note Allocator for registered devices, so that an application can offer a selection of the existing devices to the user.

Secondly, the Note Allocator provides an application level interface for requesting note channels with particular attributes. A note channel is similar in some ways to a Sound Manager sound channel; it needs to be created and disposed, and can receive various commands.

To create a note channel, the client specifies the desired polyphony and the desired tone. The Note Allocator will return a note channel that best satisfies the request. Procedural interfaces are provided to play notes and alter controller settings on the note channel.

Typically, an application will access Music Components through the Note Allocator, rather than directly.

Lastly, there is an “Instrument picker,” which provides a standard user-interface for choosing an Instrument sound.

The Note Allocator is implemented as a component. To use it, the application must find the component and open an instance of it. When that instance is closed, any note channels created with that instance are disposed.

Note Channel Allocation and Use

Note channel allocation will create a note channel by selecting and allocating a Part, within a synthesizer based on the tone requested. It also provides detailed information about an allocated note channel, and allows configuration of, and access to, external MIDI devices.

Note channel use will play individual notes, apply a specified controller to the allocated note channel, provide access to Knobs to adjust a Part's characteristics, select an Instrument based on a required tone, and modify or change the Instrument type on an existing note channel.

NANewNoteChannel

The `NANewNoteChannel` function requests a new note channel with the qualities described in the `noteRequest` structure.

```
pascal ComponentResult NANewNoteChannel
                                (NoteAllocator na,
                                NoteRequest *noteRequest,
                                NoteChannel *outChannel);
```

`na`	You obtain the Note Allocator identifier from the Components Manager's OpenComponent function. See the chapter “Component Manager” in Inside Macintosh: More Macintosh Toolbox for details.
`*noteRequest`	Attributes of note request.
`*outChannel`	New note channel handle.

```
struct NoteRequest {
   long polyphony;
   Fixed typicalPolyphony;
   ToneDescription tone;
};

struct ToneDescription {
   OSType synthesizerType;
   Str31 synthesizerName;
   Str31 instrumentName;
   long instrumentNumber;
   long gmNumber;
};
```

DESCRIPTION

The `NANewNoteChannel` function may return a value in `outChannel`, even if `noteChannel` request cannot initially be satisfied.

The Note Channel may become valid at a later time, as other Note Channels are released or other music components are registered. If an error occurs the `noteChannel` will be initialized to NIL.

ERROR CODES

none

NADisposeNoteChannel

The `NADisposeNoteChannel` function deletes the specified note channel.

```
pascal ComponentResult NADisposeNoteChannel
                                (NoteAllocator na,
                                NoteChannel noteChannel);
```

`na`	You obtain the Note Allocator identifier from the Components Manager's OpenComponent function. See the chapter “Component Manager” in Inside Macintosh: More Macintosh Toolbox for details.
`noteChannel`	Note channel to be disposed.

ERROR CODES

```
illegalNoteChannelErr
```

NAGetNoteChannelInfo

The `NAGetNoteChannelInfo` function returns the index of the Music Component for the allocated channel.

```
pascal ComponentResult NAGetNoteChannelInfo
                                (NoteAllocator na,
                                NoteChannel noteChannel,
                                long *index,
                                long *part)
```

`na`	You obtain the Note Allocator identifier from the Components Manager's OpenComponent function. See the chapter “Component Manager” in Inside Macintosh: More Macintosh Toolbox for details.
`noteChannel`	Note channel to get info on.
`*index`	Music component index.
`*part`	Music component Part pointer.

DESCRIPTION

The`NAGetNoteChannelInfo` function allows direct access to the Music Component allocated to the note channel by the Note Allocator. The index will be invalid if music components are subsequently registered or unregistered (refer to the General Event used to initially install the Music Component).

ERROR CODES

`illegalNoteChannelErr`

NAUseDefaultMIDIInput

The `NAUseDefaultMIDIInput` function defines an entry point to service external MIDI device events.

```
pascal ComponentResult NAUseDefaultMIDIInput
                                (NoteAllocator na,
                                MusicMIDIReadHookUPP readHook,
                                long refCon,
                                unsigned long flags)
```

`na`	You obtain the Note Allocator identifier from the Components Manager's OpenComponent function. See the chapter “Component Manager” in Inside Macintosh: More Macintosh Toolbox for details.
`readHook`	Process pointer for MIDI service.
`refcon`	Contains a reference constant value. The Movie Toolbox passes this reference constant to your error-notification function each time it calls your function.
`flags`	Must contain zero.

DESCRIPTION

The `NAUseDefaultMIDIInput` function specifies an application's procedure to service external MIDI events. The specified application's procedure call, defined by `readHook`, will be called when the external default MIDI device has incoming MIDI data for the application.

ERROR CODES

`midiManagerAbsentErr`

NALoseDefaultMIDIInput

The `NALoseDefaultMIDIInput` function removes the external default MIDI service procedure call, if previously defined by NAUseDefaultMIDIInput.

```
pascal ComponentResult NALoseDefaultMIDIInput
                                (NoteAllocator na);
```

`na`	You obtain the Note Allocator identifier from the Components Manager's OpenComponent function. See the chapter “Component Manager” in Inside Macintosh: More Macintosh Toolbox for details.

ERROR CODES

-1	Returned if default MIDI was not in use.

NAPrerollNoteChannel

The `NAPrerollNoteChannel` function attempts to reallocate the note channel, if it was invalid previously.

```
pascal ComponentResult NAPrerollNoteChannel
                                (NoteAllocator na,
                                NoteChannel noteChannel);
```

`na`	You obtain the Note Allocator identifier from the Components Manager's OpenComponent function. See the chapter “Component Manager” in Inside Macintosh: More Macintosh Toolbox for details.
`noteChannel`	Note channel to be re-allocated.

DESCRIPTION

The `NAPrerollNoteChannel` function attempts to reallocate the note channel, if it was invalid previously. It could have been invalid if there were no available voices on any registered music components when the note channel was created.

ERROR CODES

`illegalNoteChannelErr`
`noteChannelNotAllocatedErr`
`MusicComponent` errors for `FindTone`, `SetInstrumentNumber`

NAUnrollNoteChannel

The `NAUnrollNoteChannel` function marks a note channel as available to be stolen.

```
pascal ComponentResult NAUnrollNoteChannel
                                (NoteAllocator na,
                                NoteChannel noteChannel);
```

`na`	You obtain the Note Allocator identifier from the Components Manager's OpenComponent function. See the chapter “Component Manager” in Inside Macintosh: More Macintosh Toolbox for details.
`noteChannel`	Note channel to be unrolled.

DESCRIPTION

The MIDI channel it resides on, and the synthesizer used to play it, might be stolen by another note channel. As an example, a document whose window is moved to the background might courteously unroll its note channels.

ERROR CODES

`illegalNoteChannelErr`

NAEngageNoteChannel

The `NAEngageNoteChannel` function enables or engages the specified note channel.

```
pascal ComponentResult NAEngageNoteChannel
                                (NoteAllocator na,
                                NoteChannel noteChannel);
```

`na`	You obtain the Note Allocator identifier from the Components Manager's OpenComponent function. See the chapter “Component Manager” in Inside Macintosh: More Macintosh Toolbox for details.
`noteChannel`	Note channel to engage.

DESCRIPTION

The `NAEngageNoteChannel` function engages the specified note channel if it is currently disengaged. Any difference in notes or controllers between the engaged state and the disengaged state are sent to the music component. A note channel is engaged by default.

ERROR CODES

`illegalNoteChannelErr`

NADisengageNoteChannel

The `NADisengageNoteChannel` function causes a note channel to ignore incoming note and controller commands.

```
pascal ComponentResult NADisengageNoteChannel
                                (NoteAllocator na,
                                NoteChannel noteChannel,
                                long silenceNotes);
```

`na`	You obtain the Note Allocator identifier from the Components Manager's OpenComponent function. See the chapter “Component Manager” in Inside Macintosh: More Macintosh Toolbox for details.
`noteChannel`	Note channel to disengage.
`silenceNotes`	Silences currently playing notes. If `silenceNotes` is 1, then any notes currently playing are silenced.

DESCRIPTION

The `NADisengageNoteChannel` function is useful for fast-forwarding or rewinding to a specific spot in a score.

While the note channel is disengaged, the state of notes and controllers is still monitored, so that when the channel is engaged, the notes and controllers will be playing as if the note channel had been continuously engaged.

ERROR CODES

`illegalNoteChannelErr`

NAResetNoteChannel

The `NAResetNoteChannel` function turns “off” all currently “on” notes on the note channel, and resets all controllers to their default values.

```
pascal ComponentResult NAResetNoteChannel
                                (NoteAllocator na,
                                NoteChannel noteChannel);
```

`na`	You obtain the Note Allocator identifier from the Components Manager's OpenComponent function. See the chapter “Component Manager” in Inside Macintosh: More Macintosh Toolbox for details.
`noteChannel`	Specified note channel to reset.

DESCRIPTION

The `NAResetNoteChannel` function resets the specified note channel by turning “off” any note currently playing. Any controller applied to the note channel is also reset to its default state. The effects of the `NAResetNoteChannel` call are propagated down to the allocated Part within the appropriate Music Component.

ERROR CODES

`illegalNoteChannelErr`
errors from `MusicResetPart()`

NASetNoteChannelVolume

The `NASetNoteChannelVolume` function sets the volume on the specified note channel.

```
pascal ComponentResult NASetNoteChannelVolume
                                (NoteAllocator na,
                                NoteChannel noteChannel,
                                Fixed volume);
```

`na`	You obtain the Note Allocator identifier from the Components Manager's OpenComponent function. See the chapter “Component Manager” in Inside Macintosh: More Macintosh Toolbox for details.
`noteChannel`	Specified note channel to reset.
`volume`	16.16 volume value.

DESCRIPTION

The `NASetNoteChannelVolume` function sets the volume for the entire note channel, which is different than a controller 7 (volume controller) setting.

Both volume settings allow fractional values of 0.0 to 1.0. Each value will modify the other. Example: controller set to .5 and NASetNoteChannelVolume of .5 would result in a .25 volume level.

ERROR CODES

`illegalNoteChannelErr`

NAPlayNote

The `NAPlayNote` function plays a musical note on the specified note channel with a particular pitch and velocity.

```
pascal ComponentResult NAPlayNote
                                (NoteAllocator na,
                                NoteChannel noteChannel,
                                long pitch, long velocity);
```

`na`	You obtain the Note Allocator identifier from the Components Manager's OpenComponent function. See the chapter “Component Manager” in Inside Macintosh: More Macintosh Toolbox for details.
`noteChannel`	Specific note channel to play note.
`pitch`	0-127 where 60 is middle C. 256 (0x1.00) to 32767 (0x7f.00) are fixed PT values.
`velocity`	Value of 0 = silence.

DESCRIPTION

The `NAPlayNote` function plays a specific note. If the pitch is a number from 0 to 127, then it is the MIDI pitch, where 60 is middle-C. If the pitch is a positive number above 65535, then the value is a fixed point pitch value. Thus, microtonal values may be specified. The range 256 (0x01.00) through 32767 (0x7f.00), and all negative values, are not defined, and should not be used.

The velocity refers to how hard the key was struck (if performed on a keyboard-instrument), typically this translates directly to volume, but on many synthesizers this also subtly alters the timbre of the tone.

ERROR CODES

`illegalNoteChannelErr`

NASetController

The `NASetController` function changes the specified controller on the note channel to a particular value.

```
pascal ComponentResult NASetController
                              (NoteAllocator na, NoteChannel
                              noteChannel,
                              short controllerNumber, short
                              controllerValue);
```

`na` You obtain the Note Allocator identifier from the Components Manager's OpenComponent function. See the chapter “Component Manager” in Inside Macintosh: More Macintosh Toolbox for details.

`noteChannel` Note channel on which to change controller.

`controllerNumber`
Which controller.

`controllerValue`
Value for controller.
All controllers are reserved for use by Apple.

```
enum MusicControllers {
   kControllerModulationWheel = 1,
   kControllerBreath          = 2,
   kControllerFoot            = 4,
   kControllerPortamentoTime  = 5,
   kControllerVolume          = 7,
   kControllerBalance         = 8,
   kControllerPan             = 10,
   kControllerExpression      = 11,
   kControllerPitchBend       = 32, /* Apple unique */
   kControllerAfterTouch      = 33, /* Apple unique */
   kControllerSustain         = 64,
   kControllerPortamento      = 65,
   kControllerSostenuto       = 66,
   kControllerSoftPedal       = 67,
   kControllerReverb          = 91,
   kControllerTremolo         = 92,
   kControllerChorus          = 93,
   kControllerCeleste         = 94,
   kControllerPhaser          = 95
};
```

ERROR CODES

```
illegalNoteChannelErr
illegalControllerErr
```

NASetKnob

The `NASetKnob` function sets a particular Knob, on the specified note channel, to a particular value.

```
pascal ComponentResult NASetKnob
                                (NoteAllocator na,
                                NoteChannel noteChannel,
                                long knobNumber,
                                long knobValue)
```

`na`	You obtain the Note Allocator identifier from the Components Manager's OpenComponent function. See the chapter “Component Manager” in Inside Macintosh: More Macintosh Toolbox for details.
`noteChannel`	Note channel on which to change knob.
`knobNumber`	Knob to be set.
`knobValue`	Knob value to be set.

ERROR CODES

```
illegalNoteChannelErr
illegalKnobErr
illegalKnobValueErr
```

NAFindNoteChannelTone

The `NAFindNoteChannelTone` function locates the best fitting Instrument number on the note channel for the `toneDescription` requested.

```
pascal ComponentResult NAFindNoteChannelTone
                                (NoteAllocator na,
                                NoteChannel noteChannel,
                                ToneDescription *td,
                                long *instrumentNumber);
```

`na`	You obtain the Note Allocator identifier from the Components Manager's OpenComponent function. See the chapter “Component Manager” in Inside Macintosh: More Macintosh Toolbox for details.
`noteChannel`	Note channel to search for fit.
`*td`	Description for Instrument fit.
`*instrumentNumber`	Instrument index of fit.

```
struct ToneDescription {
   OSType synthesizerType;
   Str31 synthesizerName;
   Str31 instrumentName;
   long instrumentNumber;
   long gmNumber;
};
```

ERROR CODES

```
illegalNoteChannelErr
illegalControllerErr
```

NASetNoteChannelInstrument

The `NASetNoteChannelInstrument` function changes the Instrument setting on the note channel to the Instrument requested.

```
pascal ComponentResult NASetNoteChannelInstrument
                                (NoteAllocator na,
                                NoteChannel noteChannel,
                                short instrumentNumber);
```

`na`	You obtain the Note Allocator identifier from the Components Manager's OpenComponent function. See the chapter “Component Manager” in Inside Macintosh: More Macintosh Toolbox for details.
`noteChannel`	Note channel to apply Instrument.
`instrumentNumber`	Instrument number to apply.

ERROR CODES

`illegalNoteChannelErr`
Errors from `MusicSetInstrumentNumber()`

Miscellaneous Interface Tools

The miscellaneous interface tools provide a set of user interface dialogs to select individual Instruments, select Instruments within an arrangement and to provide copyright information for a particular Instrument.

NAPickInstrument

The `NAPickInstrument` function presents a user interface for picking an Instrument.

```
pascal ComponentResult NAPickInstrument
                                (NoteAllocator na,
                                ModalFilterUPP filterProc,
                                StringPtr prompt,
                                ToneDescription *sd,
                                unsigned long flags,
                                long refCon, Ptr *reserved1,
                                long *reserved2)
```

`na` — You obtain the Note Allocator identifier from the Components Manager's OpenComponent function. See the chapter “Component Manager” in Inside Macintosh: More Macintosh Toolbox for details.

`filterProc` — Standard modal filter upp*.

`prompt` — Dialog box prompt “New Instrument..”.

`*sd` — Tone description initialized by pick.

`flags` — Dialog flags to limit user options. Refer to list below.

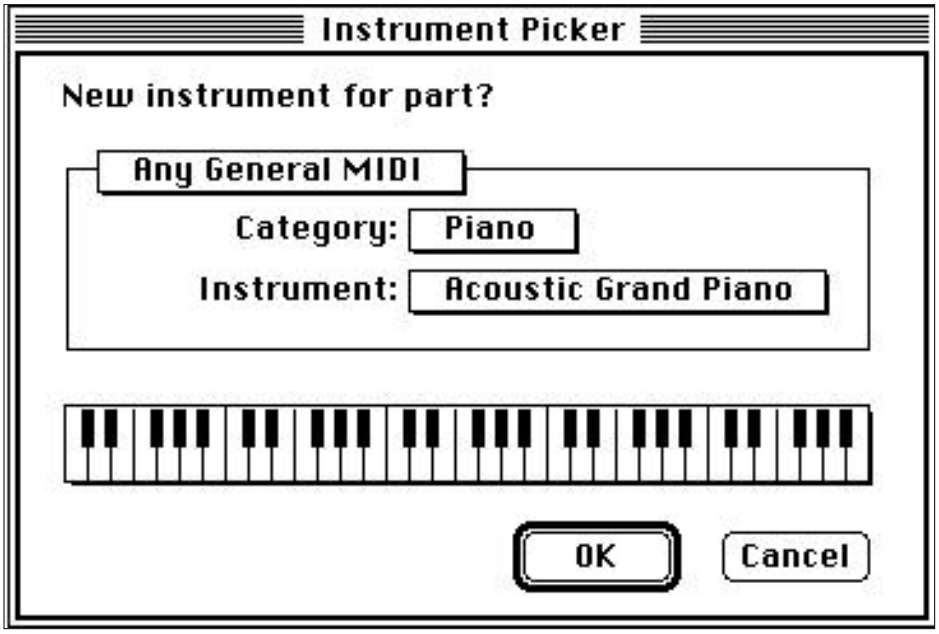
`kPickDontMix`	Don't show Drum.
`kPickSameSynth`	Don't allow options to other synths.

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

`*reserved1` — Must contained zero.

`*reserved2` — Must contained zero.

```
struct ToneDescription {
   OSType synthesizerType;
   Str31 synthesizerName;
   Str31 instrumentName;
   long instrumentNumber;
   long gmNumber;
};
```



DESCRIPTION

The two flag values limit user options displayed within the dialog box. `kPickDontMix` will not display a mix types of synthesizer types. For example, if the current synthesizer is a Drum, the `kPickDontMix` flag will display only available Drum Parts.

The `kPickSameSynth` will allow selections only within the current synthesizer.

ERROR CODES

-1	Problem opening dialog.

NAStuffToneDescription

The `NAStuffToneDescription` function initializes the tone description structure with the details of the note channel specified by the gmNumber.

```
pascal ComponentResult NAStuffToneDescription
                                (NoteAllocator na,
                                long gmNumber,
                                ToneDescription *td)
```

`na`	You obtain the Note Allocator identifier from the Components Manager's OpenComponent function. See the chapter “Component Manager” in Inside Macintosh: More Macintosh Toolbox for details.
`gmNumber`	Instrument number.
`*td`	Tone description to be stuffed.

ERROR CODES

Errors from `MusicGetInstrumentNames` and calls to General MIDI Music Component.

NAPickArrangement

The `NAPickArrangement` function displays a dialog to allow Instrument selection.

```
pascal ComponentResult NAPickArrangement
                                (NoteAllocator na,
                                ModalFilterUPP filterProc,
                                StringPtr prompt,
                                long partCount,
                                NoteRequest *noteRequestList,
                                Track t,
                                StringPtr songName)
```

`na`	You obtain the Note Allocator identifier from the Components Manager's OpenComponent function. See the chapter “Component Manager” in Inside Macintosh: More Macintosh Toolbox for details.
`filterProc`	Standard modal filter upp*.
`prompt`	Dialog box prompt.
`partCount`	Instrument selection count**.**
`*noteRequestList`	List of Instruments for selection.
`t`	Arrangement track number.
`songName`	Human readable string name displayed in dialog.

```
struct NoteRequest {
   long polyphony;
   Fixed typicalPolyphony;
   ToneDescription tone;
};
```

ERROR CODES

-1	Problem opening dialog.

NACopyrightDialog

The `NACopyrightDialog` function displays a copyright dialog with information specific to a music device.

```
pascal ComponentResult NACopyrightDialog
                                (NoteAllocator na,
                                PicHandle p, StringPtr author,
                                StringPtr copyright,
                                StringPtr other,
                                StringPtr title,
                                ModalFilterUPP filterProc,
                                long refCon)
```

`na`	You obtain the Note Allocator identifier from the Components Manager's OpenComponent function. See the chapter “Component Manager” in Inside Macintosh: More Macintosh Toolbox for details.
`p`	Picture image resource handle for dialog.
`author`	Author information.
`copyright`	Copyright information.
`other`	Any additional information.
`title`	Title information.
`filterProc`	Standard modal filter upp*.
`refcon`	Contains a reference constant value. The Movie Toolbox passes this reference constant to your error-notification function each time it calls your function.

ERROR CODES

-1	Problem opening dialog.

System Configuration

System configuration provide calls which create and maintain a database of Music Components, save configuration information in a “Music Preferences” file and establish connections to external MIDI devices.

NARegisterMusicDevice

The `NARegisterMusicDevice` function registers a music component with the Note Allocator.

```
pascal ComponentResult NARegisterMusicDevice
                    (NoteAllocator na,
                    unsigned long synthType,
                    Str31 name,
                    SynthesizerConnections *connections);
```

`na`	You obtain the Note Allocator identifier from the Components Manager's OpenComponent function. See the chapter “Component Manager” in Inside Macintosh: More Macintosh Toolbox for details.
`synthType`	Subtype of the music component.
`name`	Human readable synthesizer name.
`*connections`	MIDI connection structure.

```
struct SynthesizerConnections {
   OSType clientID;
   OSType inputPortID;
   OSType outputPortID;
   long MIDIChannel;
   long flags;
   long reserved[3];
};
```

DESCRIPTION

The `synthType` is the same as the music component’s subtype. The name is a means of distinguishing multiple instances of the same type of device. The `name` parameter is also a human readable version of the synthesizer name. If the `synthName` is not passed, the name defaults to the name of the music component type. The name will also appear in the Instrument picker dialog.

The `connections` parameter specifies the hardware connections to the device.

The `clientID`, `inputPortID` and `outputPortID` are MIDI manager identifiers. The `MIDIChannel` is the MIDI system channel value. The `flags` and `reserved` values must be zero.

ERROR CODES

`SynthesizerErr`	If too many synths registered.
`midiManagerAbsentErr`	If MIDI not available.

NAUnregisterMusicDevice

The `NAUnregisterMusicDevice` function removes a previously registered music component from the Note Allocator.

```
pascal ComponentResult NAUnregisterMusicDevice
                  (NoteAllocator na,
                  unsigned long synthType,
                  SynthesizerConnections *connections);
```

`na`	You obtain the Note Allocator identifier from the Components Manager's OpenComponent function. See the chapter “Component Manager” in Inside Macintosh: More Macintosh Toolbox for details.
`synthType`	Synthesizer type string.
`*connections`	MIDI connection structure.

ERROR CODES

`NoteAllocator` errors from `NAResetNoteChannel`

errors from `CloseComponent`

NAGetRegisteredMusicDevice

The `NAGetRegisteredMusicDevice` function returns specifics about music components registered to the specified Note Allocator instance.

```
pascal ComponentResult NAGetRegisteredMusicDevice
                  (NoteAllocator na, short index,
                  unsigned long *synthType, Str31 name,
                  SynthesizerConnections *connections,
                  MusicComponent *mc);
```

`na`	You obtain the Note Allocator identifier from the Components Manager's OpenComponent function. See the chapter “Component Manager” in Inside Macintosh: More Macintosh Toolbox for details.
`index`	0 or 1 - max music components.
`*synthType`	Synthesizer type string.
`name`	Human readable synthesizer name.
`*connections`	MIDI connection structure.
`*mc`	Music component instance.

DESCRIPTION

An `index` value of zero will cause `NAGetRegisteredMusicDevice` to return a total count of registered music components. An index value of 1 through the maximum number of music components will return information about the music component specified by the index.

The music component information returned by this call provides direct access to the particular music component. Refer to the function calls in the Music Component Interface section for additional details.

ERROR CODES

none

NAGetDefaultMIDIInput

The `NASGetDefaultMIDIInput` function is used to obtain external MIDI connection information.

```
pascal ComponentResult NAGetDefaultMIDIInput
                                (NoteAllocator na,
                                SynthesizerConnections *sc);
```

`na` — You obtain the Note Allocator identifier from the Components Manager's OpenComponent function. See the chapter “Component Manager” in Inside Macintosh: More Macintosh Toolbox for details.

`*sc` — MIDI connection structure.

```
struct SynthesizerConnections {
   OSType clientID;
   OSType inputPortID;
   OSType outputPortID;
   long MIDIChannel;
   long flags;
   long reserved[3];
};
```

DESCRIPTION

The `NASGetDefaultMIDIInput` function returns an initialized SynthesizerConnections structure containing information about any default external MIDI device attached to the system. The external MIDI device provides note input directly to the Note Allocator.

ERROR CODES

none

NASetDefaultMIDIInput

The `NASetDefaultMIDIInput` function initializes an external MIDI device used to receive external note input.

```
pascal ComponentResult NASetDefaultMIDIInput
                                (NoteAllocator na,
                                SynthesizerConnections *sc);
```

`na`	You obtain the Note Allocator identifier from the Components Manager's OpenComponent function. See the chapter “Component Manager” in Inside Macintosh: More Macintosh Toolbox for details.
`*sc`	MIDI connection structure.

DESCRIPTION

The `SynthesizerConnections` structure members `clientID`, `inputPortID` and `outputPortID` (described in above) are MIDI manager identifiers. The `MIDIChannel` is the MIDI system channel value. The `flags` and `reserved` values must be zero.

ERROR CODES

none

NASaveMusicConfiguration

The `NASaveMusicConfiguration` saves the current list of registered devices to a file.

```
pascal ComponentResult NASaveMusicConfiguration
                                (NoteAllocator na);
```

`na`	You obtain the Note Allocator identifier from the Components Manager's OpenComponent function. See the chapter “Component Manager” in Inside Macintosh: More Macintosh Toolbox for details.

DESCRIPTION

The `NASaveMusicConfiguration` saves the current list of registered devices to a file. This file is read whenever a Note Allocator connection is opened, restoring the previously configured list of devices. The file is called “Music Preferences” and is placed in the “Preferences” subfolder of the system folder.

ERROR CODES

-1	Returned if problem opening or creating the Music Preferences file in the system folder.

Music Component Interface

The Music Components are not usually called directly unless an application is required to access the music device directly. This is achieved by first allocating a noteChannel. By using `NAGetNoteChannelInfo()` and `NAGetRegisteredMusicDevice()`, the application can locate the specific music component and Part number.

This layer is of interest to application developers who wish to access low-level functionality of synthesizers and for developers of synthesizers (nubus cards, MIDI devices or software algorithms) who wish to make the capabilities of their synthesizers available to QuickTime.

Synthesizer Access

Music Component synthesizer access provides services to obtain specific information about the current synthesizer and obtain a best Instrument fit for a requested tone from the available Instruments within the synthesizer. The synthesizer access can also play a note with a specified pitch, volume and duration, get and set a particular synthesizer Knob, obtain default synthesizer Knob information and get and set external MIDI procedure name entry points.

MusicGetDescription

The `MusicGetDescription` function returns a structure describing the synthesizer controlled by the Music Component device.

```
pascal ComponentResult MusicGetDescription
                              (MusicComponent mc,
                              SynthesizerDescription *sd);
```

`mc` Music component instance returned by NAGetRegisteredMusicDevice().

`*sd` Pointer to synthesizer description.

```
struct SynthesizerDescription {
   OSType           type;
   Str31            name;
   unsigned long    flags;
   unsigned long    voiceCount;
   unsigned long    partCount;
   unsigned long    instrumentCount;
   unsigned long    modifiableInstrumentCount;
   unsigned long    channelMask;
   unsigned long    drumPartCount;
   unsigned long    drumCount;
   unsigned long    modifiableDrumCount;
   unsigned long    drumChannelMask;
   unsigned long    outputCount;
   unsigned long    latency;
   unsigned long    controllers[4];
   unsigned long    gmInstruments[4];
   unsigned long    gmDrums[4];
};
```

DESCRIPTION

The `MusicGetDescription` function returns a structure describing the specified music component device. The `SynthesizerDescription` record is filled out by the particular music component.

ERROR CODES

`synthesizerErr` A synthesizer specific error has occurred.

`cantSendToSynthesizerErr`

The component is unable to send commands to the synthesizer, for example if the MusicSetMIDIProc() routine has not been called.

MusicFindTone

The `MusicFindTone` function returns an Instrument number based on a tone description.

```
pascal ComponentResult MusicFindTone
                                (MusicComponent mc,
                                ToneDescription *td,
                                long *instrumentNumber,
                                long *fit);
```

`mc` Music component instance returned by NAGetRegisteredMusicDevice().

`*td` Pointer to a tone description.

`*instrumentNumber`
Instrument number of match.

`*fit` Returns the fit quality.

`kInstrumentMatchSynthesizerType`

The requested synthesizer type was found.

`kInstrumentMatchSynthesizerName`

The particular instance of the synthesizer requested was found.

`kInstrumentMatchName`

The toneDescription's Instrument name matched an appropriate Instrument on the synthesizer.

`kInstrumentMatchNumber`

The toneDescription's Instrument number matched an appropriate Instrument on the synthesizer.

`kInstrumentMatchGMNumber`

The General MIDI equivalent was used to find an appropriate Instrument on the synthesizer.

```
typedef struct
{
   OSType synthesizerType; /* component subtype */
   Str31 synthesizerName;  /* instants name of synth */
   Str31 instrumentName;   /* human use name */
   long instrumentNumber;  /* instrument # if synth-type
                           matches */
   long gmNumber;          /* Best matching general MIDI
                           number */
} ToneDescription;
```

DESCRIPTION

The `MusicFindTone` function returns the best-matching Instrument number for this device. How close a match was attained is returned in “fit”.

The Music component should search in the following order:

1 If the synthesizer is a general MIDI device, use the `gmNumber`.

2 If `synthesizerType` matches, first try to match `instrumentName`, else try `instrumentNumber`. Failing that, try the `gmNumber`.

3 If `synthesizerType` doesn’t match, try the `instrumentName`, then the Instrument number.

If none of these rules apply, or the fields are “blank” (zero for the type or numeric fields, or zero-length for the strings) then the call return Instrument 1 and a fit value of zero. The synthesizerName field may be ignored by the component; it is used by the Note Allocator when deciding which music device to use.

ERROR CODES

`synthesizerErr` A synthesizer specific error has occurred.

`cantSendToSynthesizerErr`

The component is unable to send commands to the synthesizer, for example if the MusicSetMIDIProc() routine has not been called.

`illegalInstrumentErr`

The Instrument number is out of valid range.

MusicPlayNote

The `MusicPlayNote` function plays a specific note on the specified Part characterized by its pitch and velocity.

```
pascal ComponentResult MusicPlayNote
                                (MusicComponent mc, long part,
                                long pitch,
                                long velocity);
```

`mc`	Music component instance returned by NAGetRegisteredMusicDevice().
`part`	Part number to apply controller.
`pitch`	0-127 MIDI pitch. > 65535 microtonal.
`velocity`	0-127 where 0 = silence.

DESCRIPTION

The `MusicPlayNote` function is used to play notes with their pitch, if MIDI, specified by a number from 0 to 127, if a MIDI pitch, where 60 is middle-C. If the pitch is a positive number above 65535, then the value is a fixed point pitch value. Thus, microtonal values may be specified. The range 256 (0x01.00) through 32767 (0x7f.00), and all negative values, are not defined, and should not be used.

Velocity refers to how hard the key is struck (if performed on a keyboard-Instrument), typically this translates directly to volume, but on many synthesizers this also subtly alters the timbre of the tone.

The current note continues to play until a `MusicPlayNote()` with the same pitch and velocity of 0 turns the note off.

ERROR CODES

`synthesizerErr`	A synthesizer specific error has occurred.
`illegalPartErr`	A Part number outside the valid range (1..partCount) has been passed.
`cantSendToSynthesizerErr`	The component is unable to send commands to the synthesizer, for example if the MusicSetMIDIProc() routine has not been called.

MusicGetKnob

The `MusicGetKnob` function returns the value of the specified synthesizer Knob.

```
pascal ComponentResult MusicGetKnob
                                (MusicComponent mc,
                                long knobNumber);
```

`mc` Music component instance.

`knobNumber` Instrument Knob number.

DESCRIPTION

The Knob controls an aspect of the entire synthesizer, not limited or specific to a Part or Instrument within the synthesizer.

ERROR CODES

`synthesizerErr` A synthesizer specific error has occurred.

`cantSendToSynthesizerErr`

The component is unable to send commands to the synthesizer, for example if the MusicSetMIDIProc() routine has not been called.

`illegalKnobErr` A Knob number outside the valid range (1..knobCount) has been.

`illegalKnobValueErr`

The Knob value is outside its legal range, as returned in its KnobDescription.

MusicSetKnob

The `MusicSetKnob` function modifies the value of the specified synthesizer Knob.

```
pascal ComponentResult MusicSetKnob
                                (MusicComponent mc,
                                long knobNumber,
                                long knobValue);
```

`mc`	Music component instance.
`knobNumber`	Instrument Knob number.
`knobValue`	Value for specified Knob.

DESCRIPTION

The Knob controls an aspect of the entire synthesizer, not limited or specific to a Part or Instrument within the synthesizer.

ERROR CODES

`synthesizerErr`	A synthesizer specific error has occurred.
`cantSendToSynthesizerErr`	The component is unable to send commands to the synthesizer, for example if the MusicSetMIDIProc() routine has not been called.
`illegalKnobErr`	A Knob number outside the valid range (1..knobCount) has been.
`illegalKnobValueErr`	The Knob value is outside its legal range, as returned in its KnobDescription.

MusicGetKnobDescription

The `MusicGetKnobDescription` function returns an initialized `KnobDescription` structure pointer for a synthesizer's Knob. The Knob controls an aspect of the entire synthesizer, not limited or specific to a Part or Instrument within the synthesizer.

```
pascal ComponentResult MusicGetKnobDescription
                            (MusicComponent mc,
                            long knobNumber,
                            KnobDescription *mkd);
```

`mc`	Music component instance.
`knobNumber`	Particular Knob.
`*mkd`	Pointer to KnobDescription.

DESCRIPTION

The `MusicGetKnobDescription` function will return an initialized `KnobDescription` structure pointer. This structure will provide the application default values associated with the particular Knob. This call allows the Knob to be reset to some known, usable value.

ERROR CODES

`synthesizerErr`	A synthesizer specific error has occurred.
`cantSendToSynthesizerErr`	The component is unable to send commands to the synthesizer, for example if the MusicSetMIDIProc() routine has not been called.
`illegalKnobErr`	A Knob number outside the valid range (1..knobCount) has been.
`illegalKnobValueErr`	The Knob value is outside its legal range, as returned in its KnobDescription.

MusicGetMIDIProc

The `MusicGetMIDIProc` function returns the currently active function call used to process external MIDI notes.

```
pascal ComponentResult MusicGetMIDIProc
                        (MusicComponent mc,
                        MusicMIDISendProcPtr *MIDISendProc,
                        long *refCon);
```

`mc`	Music component instance returned by NAGetRegisteredMusicDevice().
`*MIDISendProc`	Pointer into MIDI serial port call.
`*refcon`	Contains a reference constant value. The Movie Toolbox passes this reference constant to your error-notification function each time it calls your function.

DESCRIPTION

The `MusicGetMIDIProc` function returns the active `*MIDISendProc` pointer. This pointer provides a function call initialized by QMA, and provides access to an external MIDI port for serial communications. If the port is uninitialized `*MIDISendProc` will return zero.

ERROR CODES

`synthesizerErr`	A synthesizer specific error has occurred.
`cantSendToSynthesizerErr`	The component is unable to send commands to the synthesizer, for example if the MusicSetMIDIProc() routine has not been called.
`illegalChannelErr`	A MIDI channel value outside the valid range (1..16 or 0) has been passed.

MusicSetMIDIProc

The `MusicSetMIDIProc` function initializes the MIDISendProc value specifying the procedure entry point for external MIDI serial communications.

```
pascal ComponentResult MusicSetMIDIProc
                        (MusicComponent mc,
                        MusicMIDISendProcPtr MIDISendProc,
                        long refCon);
```

`mc`	Music component instance returned by NAGetRegisteredMusicDevice().
`MIDISendProc`	MIDI serial port call pointer.
`refcon`	Contains a reference constant value. The Movie Toolbox passes this reference constant to your error-notification function each time it calls your function.

ERROR CODES

`synthesizerErr`	A synthesizer specific error has occurred.
`cantSendToSynthesizerErr`	The component is unable to send commands to the synthesizer, for example if the MusicSetMIDIProc() routine has not been called.
`illegalChannelErr`	A MIDI channel value outside the valid range (1..16 or 0) has been passed.

Instrument Control

Music Component Instrument access provides services that return or initialize a specified Part to a particular Instrument, return an organized group of Instrument or Drum names available, return the Instrument number assigned to a specified Part. In addition, the Instrument access can store modified Parts into the modifiable Instrument store, get detailed information about each available Instrument, and provide detailed default settings for an Instrument's Knob settings.

MusicGetInstrument

The `MusicGetInstrument` returns a handle containing an initialized `InstrumentData` structure for the specified Part.

```
pascal ComponentResult MusicGetInstrument
                                (MusicComponent mc, long part,
                                InstrumentDataHandle *iH);
```

`mc`	Music component instance.
`part`	Instrument Part number.
`*iH`	Data handle initialized by call.

```
struct InstrumentData {
   ToneDescription  tone;
   long             knobCount;
   long             knob[1];
};

struct ToneDescription {
   OSType  synthesizerType;
   Str31   synthesizerName;
   Str31   instrumentName;
   long    instrumentNumber;
   long    gmNumber;
};
```

Note: This handle is allocated in the caller’s heap, and must be disposed by the caller.

DESCRIPTION

Instruments can be stored either to disk or in the synthesizer's User Modifiable Instrument range.

Instrument data saved to disk, for example, and restored to the synthesizer at a later time (`MusicSetInstrument`) provides a means to modify and restore Instruments between sessions.

ERROR CODES

`synthesizerErr`	A synthesizer specific error has occurred.
`illegalPartErr`	A Part number outside the valid range (1..partCount) has been passed.

`cantSendToSynthesizerErr`

The component is unable to send commands to the synthesizer, for example if the MusicSetMIDIProc() routine has not been called.

MusicSetInstrument

The `MusicSetInstrument` function initializes the specified Part on the synthesizer with the passed instrument data handle.

```
pascal ComponentResult MusicSetInstrument
                                (MusicComponent mc,long part,
                                InstrumentDataHandle iH);
```

`mc` Music component instance.

`part` Instrument Part number.

`iH` Instrument data structure.

ERROR CODES

`synthesizerErr` A synthesizer specific error has occurred.

`illegalPartErr` A Part number outside the valid range (1..partCount) has been passed.

`cantSendToSynthesizerErr`

The component is unable to send commands to the synthesizer, for example if the MusicSetMIDIProc() routine has not been called.

MusicGetInstrumentNumber

The `MusicGetInstrumentNumber` function returns the Instrument number currently assigned to that Part.

```
pascal ComponentResult MusicGetInstrumentNumber
                                (MusicComponent mc,
                                long part);
```

`mc` Music component instance.

`part`	Part number containing Instrument.

ERROR CODES

`synthesizerErr`	A synthesizer specific error has occurred.
`illegalPartErr`	A Part number outside the valid range (1..partCount) has been passed.
`cantSendToSynthesizerErr`	The component is unable to send commands to the synthesizer, for example if the MusicSetMIDIProc() routine has not been called.

MusicSetInstrumentNumber

The `MusicSetInstrumentNumber` function assigns a particular Instrument, within the specified music component, to the specified Part. The resulting Instrument number may be determined with the `MusicFindTone()`.

```
pascal ComponentResult MusicSetInstrumentNumber
                                (MusicComponent mc, long part,
                                long instrumentNumber);
```

`mc` Music component instance.

`part` Part number to be set.

`instrumentNumber`
Instrument number used by Part.

DESCRIPTION

The Instrument number, resulting from the `MusicSetInstrumentNumber` function call, can be determined with `MusicFindTone()` call.

ERROR CODES

`synthesizerErr` A synthesizer specific error has occurred.

`illegalPartErr` A Part number outside the valid range (1..partCount) has been passed.

`cantSendToSynthesizerErr`

The component is unable to send commands to the synthesizer, for example if the MusicSetMIDIProc() routine has not been called.

`illegalInstrumentErr`

The Instrument number is out of valid range.

MusicStoreInstrument

The `MusicStoreInstrument` puts whatever Instrument is on the specified Part into the synthesizer's Instrument store.

```
pascal ComponentResult MusicStoreInstrument
                                (MusicComponent mc,long part,
                                long instrumentNumber);
```

`mc` Music component instance.

`part` Part to store Instrument.

`instrumentNumber`
Instrument number to be stored in Part.

DESCRIPTION

The `InstrumentNumber` must be between 1 and the synthesizer's `modifiableInstrumentCount`, as defined by the synthesizer description.

ERROR CODES

`synthesizerErr` A synthesizer specific error has occurred.

`illegalPartErr` A Part number outside the valid range (1..partCount) has been passed.

`cantSendToSynthesizerErr`
The component is unable to send commands to the synthesizer, for example if the MusicSetMIDIProc() routine has not been called.

`illegalInstrumentErr`
The Instrument number is out of valid range.

MusicGetInstrumentNames

The `MusicGetInstrumentNames` function returns a list of Instrument names known by the specified Music Component.

```
pascal ComponentResult MusicGetInstrumentNames
                        (MusicComponent mc,
                        long modifiableInstruments,
                        Handle *instrumentNames,
                        Handle *instrumentCategoryLasts,
                        Handle *instrumentCategoryNames)
```

`mc` Music component instance returned by NAGetRegisteredMusicDevice().

`modifiableInstruments`

Instrument count to return. A value of 0 will return only a fixed Instrument count. A value of 1 will return the fixed and user modifiable Instrument count.

`*instrumentNames`

The requested list of Instrument names formatted as a short followed by packed strings.

`*instrumentCategoryLasts`

A handle containing a group of short integers, the first of which contains the number of shorts to follow. Examples: {0},{1,20},{5,1,2,3,4,5}.

`*instrumentCategoryNames`

Instrument category names formatted as a short followed by a list of names.

DESCRIPTION

The `MusicGetInstrumentNames` function returns a list of Instruments, organized in groups. The Instrument list provides application configuration information about the specified Music Component. The information, and its format, is intended for application dialog support.

ERROR CODES

`synthesizerErr` A synthesizer specific error has occurred.

`cantSendToSynthesizerErr`

The component is unable to send commands to the synthesizer, for example if the MusicSetMIDIProc() routine has not been called.

MusicGetDrumNames

The `MusicGetDrumNames` function returns a list of Drum names for the music component. Unlike `MusicGetInstrumentNames`, which returns names grouped in categories, `MusicGetDrumNames` returns a single list containing all available Drum names.

```
pascal ComponentResult MusicGetDrumNames
                                (MusicComponent mc,
                                long modifiableInstruments,
                                Handle *instrumentNumbers,
                                Handle *instrumentNames)
```

`mc` Music component instance returned by NAGetRegisteredMusicDevice().

`modifiableInstruments`
Maximum Drum count to return.

`*instrumentNumbers`
Handle to Instrument number.

`*instrumentNames`
Handle to Instrument names.

DESCRIPTION

The `MusicGetDrumNames` function returns a single list of names. This is unlike the `MusicGetInstrumentNames` call which returns a set of named groups.

ERROR CODES

`synthesizerErr` A synthesizer specific error has occurred.

`cantSendToSynthesizerErr`

The component is unable to send commands to the synthesizer, for example if the MusicSetMIDIProc() routine has not been called.

MusicGetInstrumentAboutInfo

The `MusicGetInstrumentAboutInfo` function fills out a structure providing information about a specific Part within a particular Instrument. This is intended to provide copyright information about the synthesizer or its sounds, and may be seen by the user by clicking the “About...” button in the synthesizer picker.

```
pascal ComponentResult MusicGetInstrumentAboutInfo
                                (MusicComponent mc,long part,
                                InstrumentAboutInfo *iai);
```

`mc` Music component instance.

`part` Part number to return information.

`*iai` Pointer to instrumentAboutInfo.

```
struct InstrumentAboutInfo {
   PicHandle  p;
   Str255     author;
   Str255     copyright;
   Str255     other;
};
```

ERROR CODES

`synthesizerErr` A synthesizer specific error has occurred.

`illegalPartErr` A Part number outside the valid range (1..partCount) has been passed.

`cantSendToSynthesizerErr`

The component is unable to send commands to the synthesizer, for example if the MusicSetMIDIProc() routine has not been called.

MusicGetInstrumentKnobDescription

The `MusicGetInstrumentKnobDescription` function returns an initialized `KnobDescription` structure pointer for the specified Instrument Knob.

```
pascal ComponentResult MusicGetInstrumentKnobDescription
                                (MusicComponent mc,
                                long knobNumber,
                                KnobDescription *mkd);
```

`mc` Music component instance.

`knobNumber` Knob number to be retrieved.

`*mkd` Knob description structure pointer.

```
struct KnobDescription {
   Str31 name;
   long  lowValue;
   long  highValue;
   long  defaultValue;
   long  flags;
};
```

DESCRIPTION

The `MusicGetInstrumentKnobDescription` function's `KnobDescription` structure provides the application with low, high and default values for the specified Knob. Setting every Knob to its default value will produce a simple generic sound.

ERROR CODES

`synthesizerErr` A synthesizer specific error has occurred.

`cantSendToSynthesizerErr`

The component is unable to send commands to the synthesizer, for example if the MusicSetMIDIProc() routine has not been called.

`illegalKnobErr` A Knob number outside the valid range (1..knobCount) has been.

`illegalKnobValueErr`

The Knob value is outside its legal range, as returned in its KnobDescription.

MusicGetDrumKnobDescription

The `MusicGetDrumKnobDescription` function returns a pointer to an initialized `KnobDescription` structure for the specified Drum Knob.

```
pascal ComponentResult MusicGetDrumKnobDescription
                                (MusicComponent mc,
                                long knobNumber,
                                KnobDescription *mkd);
```

`mc`	Music component instance.
`knobNumber`	Drum's Knob number.
`*mkd`	Knob description structure.

DESCRIPTION

The `MusicGetDrumKnobDescription` returns an initialized Knob structure providing the application with default values for the specified Knob. This call allows the specific Knob values, if necessary, to be restored (reset) to a known usable state.

ERROR CODES

`synthesizerErr`	A synthesizer specific error has occurred.
`cantSendToSynthesizerErr`	The component is unable to send commands to the synthesizer, for example if the MusicSetMIDIProc() routine has not been called.
`illegalKnobErr`	A Knob number outside the valid range (1..knobCount) has been.
`illegalKnobValueErr`	The Knob value is outside its legal range, as returned in its KnobDescription.

Part Access

Music Component Part access provides services to get and set synthesizer Part parameters, get and set a Part's human interface name, get and set the value for a particular Part Knob, and to reset a specified Part to a default state and to get and apply controller values to individual Parts to modify their characteristics.

MusicGetPart

The `MusicGetPart` function returns the MIDI channel and maximum polyphony for a particular Part in the `*MIDIChannel` and `*polyphony` parameters.

```
pascal ComponentResult MusicGetPart
                                (MusicComponent mc, long part,
                                long *MIDIChannel,
                                long *polyphony)
```

`mc`	The music component.
`part`	The music component Part requested.
`*MIDIChannel`	Pointer to long for MIDIChannel result.
`*polyphony`	Pointer to long for polyphony result.

ERROR CODES

`synthesizerErr`	A synthesizer specific error has occurred.
`illegalPartErr`	A Part number outside the valid range (1..partCount) has been passed.
`cantSendToSynthesizerErr`	The component is unable to send commands to the synthesizer, for example if the MusicSetMIDIProc() routine has not been called.
`illegalChannelErr`	A MIDI channel value outside the valid range (1..16 or 0) has been passed.

MusicSetPart

The `MusicSetPart` function sets the MIDI channel and maximum polyphony for the specified Part in the `MIDIChannel` and `polyphony` parameters.

```
pascal ComponentResult MusicSetPart
                                (MusicComponent mc, long part,
                                long MIDIChannel,
                                long polyphony)
```

`mc`	Music component instance.
`part`	Part to be set.
`MIDIChannel`	The MIDI channel to be set to.
`polyphony`	The maximum voices or polyphony.

ERROR CODES

`synthesizerErr`	A synthesizer specific error has occurred.
`illegalPartErr`	A Part number outside the valid range (1..partCount) has been passed.
`illegalVoiceAllocationErr`	The Part request has exceeded the Parts available for the specific synthesizer.
`cantSendToSynthesizerErr`	The component is unable to send commands to the synthesizer, for example if the MusicSetMIDIProc() routine has not been called.
`illegalChannelErr`	A MIDI channel value outside the valid range (1..16 or 0) has been passed.

MusicGetPartName

The `MusicGetPartName` function returns the string name of the requested Part number.

```
pascal ComponentResult MusicGetPartName
                                (MusicComponent mc,long part,
                                Str31 name);
```

`mc` Music component instance.

`part` Music Part to get name.

`name` Returned music Part name.

DESCRIPTION

The name string is a human readable name used by selection dialogs or configuration information.

ERROR CODES

`synthesizerErr` A synthesizer specific error has occurred.

`illegalPartErr` A Part number outside the valid range (1..partCount) has been passed.

`cantSendToSynthesizerErr`

The component is unable to send commands to the synthesizer, for example if the MusicSetMIDIProc() routine has not been called.

MusicSetPartName

The `MusicSetPartName` function initializes the name portion of the specified Part number.

```
pascal ComponentResult MusicSetPartName
                                (MusicComponent mc,long part,
                                Str31 name);
```

`mc` Music component instance returned by NAGetRegisteredMusicDevice().

`part` Music Part to apply name.

`name` Name to apply to music Part.

DESCRIPTION

The name string is a human readable name used by selection dialogs or configuration information.

ERROR CODES

`synthesizerErr` A synthesizer specific error has occurred.

`illegalPartErr` A Part number outside the valid range (1..partCount) has been passed.

`cantSendToSynthesizerErr`

The component is unable to send commands to the synthesizer, for example if the MusicSetMIDIProc() routine has not been called.

MusicGetPartKnob

The `MusicGetPartKnob` function gets the current value of the specified Part Knob.

```
pascal ComponentResult MusicGetPartKnob
                                (MusicComponent mc,long part,
                                long knobNumber);
```

`mc` Music component instance.

`part` The Part number.

`knobNumber` The Part Knob number.

ERROR CODES

`synthesizerErr` A synthesizer specific error has occurred.

`illegalPartErr` A Part number outside the valid range (1..partCount) has been passed.

`cantSendToSynthesizerErr`

The component is unable to send commands to the synthesizer, for example if the MusicSetMIDIProc() routine has not been called.

`illegalKnobErr` A Knob number outside the valid range (1..knobCount) has been.

`illegalKnobValueErr`

The Knob value is outside its legal range, as returned in its KnobDescription.

MusicSetPartKnob

The `MusicSetPartKnob` function sets the specified Part Knob to the value of `KnobValue`.

```
pascal ComponentResult MusicSetPartKnob
                                (MusicComponent mc,long part,
                                long knobNumber,
                                long knobValue);
```

`mc`	Music component instance.
`part`	The Part number.
`knobNumber`	The Part Knob number to be set.
`knobValue`	The new Part Knob value.

ERROR CODES

`synthesizerErr`	A synthesizer specific error has occurred.
`illegalPartErr`	A Part number outside the valid range (1..partCount) has been passed.
`cantSendToSynthesizerErr`	The component is unable to send commands to the synthesizer, for example if the MusicSetMIDIProc() routine has not been called.
`illegalKnobErr`	A Knob number outside the valid range (1..knobCount) has been.
`illegalKnobValueErr`	The Knob value is outside its legal range, as returned in its KnobDescription.

MusicResetPart

The `MusicResetPart` function silences all sounds on the specified Part, and resets all controllers to their default values. The default value for all controllers is 0 (zero), except volume. Volume is set to its maximum 32767 or, in hexadecimal, 7F.FF.

```
pascal ComponentResult MusicResetPart
                                (MusicComponent mc,
                                long Part);
```

`mc`	Music component instance returned by NAGetRegisteredMusicDevice().
`part`	Part number to apply controller.

ERROR CODES

`synthesizerErr`	A synthesizer specific error has occurred.
`illegalPartErr`	A Part number outside the valid range (1..partCount) has been passed.
`cantSendToSynthesizerErr`	The component is unable to send commands to the synthesizer, for example if the MusicSetMIDIProc() routine has not been called.

MusicGetController

The `MusicGetController` function returns the value of the specified controller on the specified Part.

```
pascal ComponentResult MusicGetController
                                (MusicComponent mc, long part,
                                long controllerNumber);
```

`mc`	Music component instance returned by NAGetRegisteredMusicDevice().
`part`	Part number to apply controller.
`contollerNumber`	Controller number.

ERROR CODES

`synthesizerErr`	A synthesizer specific error has occurred.
`illegalPartErr`	A Part number outside the valid range (1..partCount) has been passed.
`cantSendToSynthesizerErr`	The component is unable to send commands to the synthesizer, for example if the MusicSetMIDIProc() routine has not been called.
`illegalControllerErr`	The controller number is either out of the legal range 1 through 128, or is not recognized by this particular component.

MusicSetController

The `MusicSetController` function initializes the value of the specified controller on the specified Part.

```
pascal ComponentResult MusicSetController
                                (MusicComponent mc, long part,
                                long controllerNumber,
                                long controllerValue);
```

`mc` Music component instance returned by NAGetRegisteredMusicDevice().

`part` Part number to apply controller.

`contollerNumber`
Controller number.

`controllerValue`
Value for controller.

Controllers 0 through 127 correspond roughly to the standard MIDI controllers. The value is always a signed 16 bit number where the lower 8 bits are fractional. The range is -7F.FF through +7F.FF, or -32767 to 32767, or 0x8001 to 0x7FFF.

Controller 32 is pitch bend, and it is defined to be in semitones, where the lower 8 bits specify 256ths of a semitone.

ERROR CODES

`synthesizerErr` A synthesizer specific error has occurred.

`illegalPartErr` A Part number outside the valid range (1..partCount) has been passed.

`synthesizerErr` A synthesizer specific error has occurred.

`cantSendToSynthesizerErr`
The component is unable to send commands to the synthesizer, for example if the MusicSetMIDIProc() routine has not been called.

`illegalControllerErr`

The controller number is either out of the legal range 1 through 128, or is not recognized by this particular component.

Synthesizer Timing

Music component synthesizer timing provides services to get and modify the master timer reference used by the synthesizer.

MusicGetMasterTune

The `MusicGetMasterTune` function returns the master reference timer which is used as the base time clock.

```
pascal ComponentResult MusicGetMasterTune
                                (MusicComponent mc);
```

`mc` Music component instance returned by NAGetRegisteredMusicDevice().

ERROR CODES

`synthesizerErr` A synthesizer specific error has occurred.

`cantSendToSynthesizerErr`

The component is unable to send commands to the synthesizer, for example if the MusicSetMIDIProc() routine has not been called.

MusicSetMasterTune

The `MusicSetMasterTune` function alters the master reference timer which is used as the base time clock.

```
pascal ComponentResult MusicSetMasterTune
                                (MusicComponent mc,
                                Fixed masterTune);
```

`mc` Music component instance returned by NAGetRegisteredMusicDevice().

`masterTune` A fixed 16.16 number allowing shifts by fractional values.

ERROR CODES

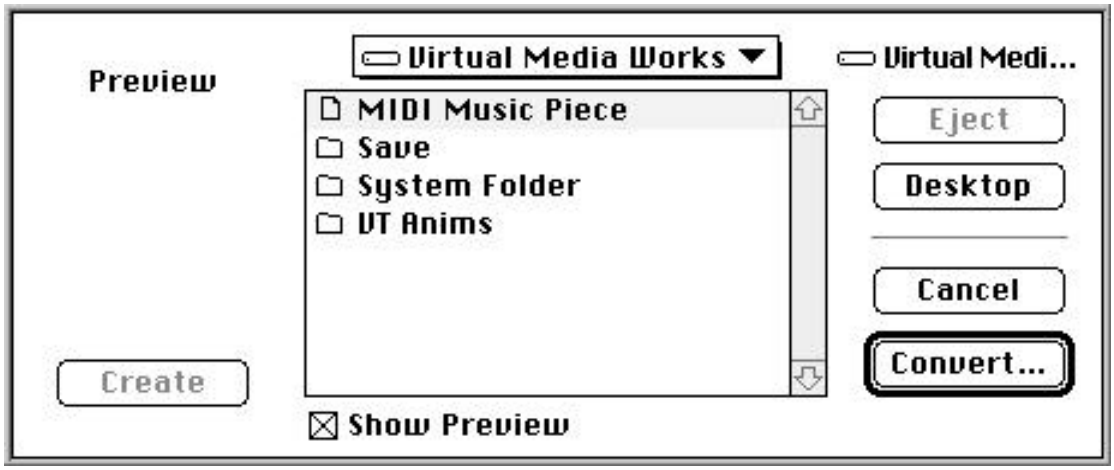
`synthesizerErr` A synthesizer specific error has occurred.

`cantSendToSynthesizerErr`

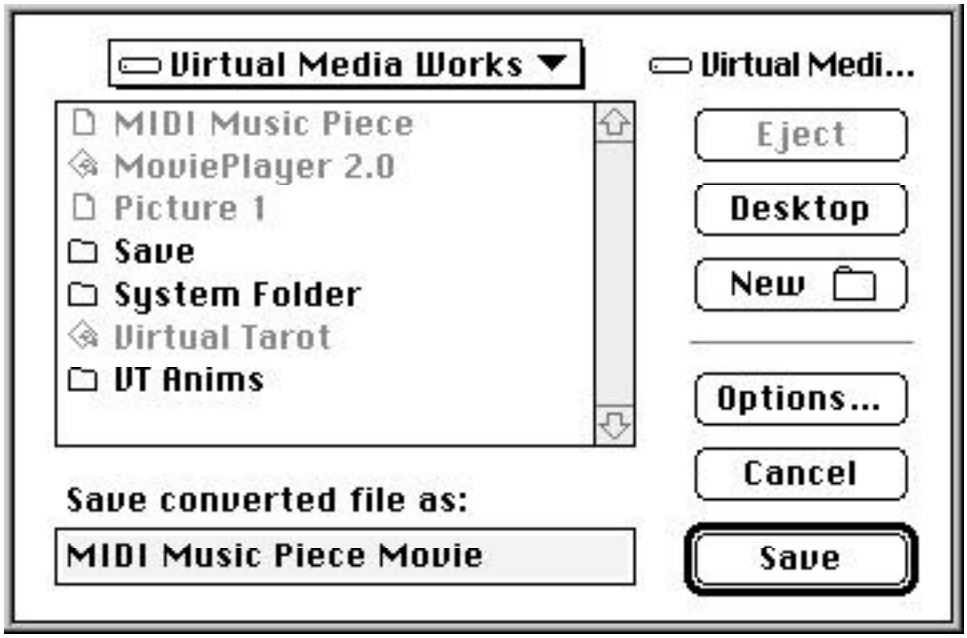
The component is unable to send commands to the synthesizer, for example if the MusicSetMIDIProc() routine has not been called.

CONVERSION OF STANDARD MIDI

MoviePlayer 2.0 allows you to open and select a standard Macintosh MIDI file. Once selected the open button will change to Convert.



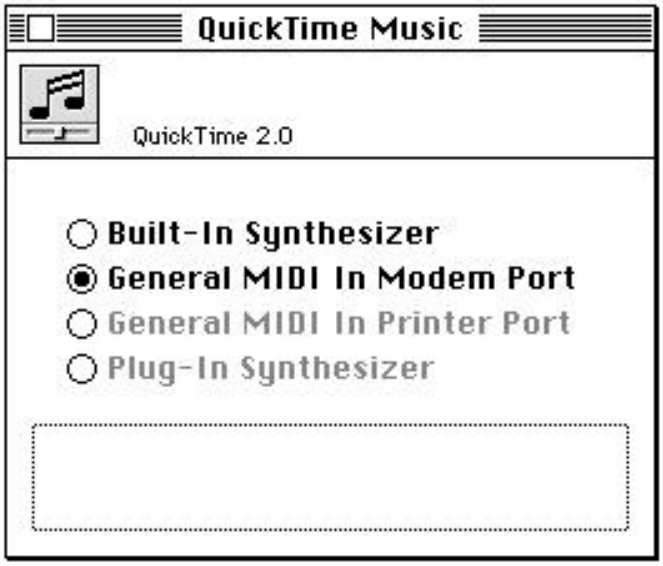
After the file is converted, MoviePlayer will prompt to save the converted file with the suffix movie.



Once saved, a named QuickTime movie controller is displayed and the converted MIDI file can be played.

MUSIC CONFIGURATION UTILITY

The following illustration is a preliminary version of the user configuration utility. The printer port does not appear because LocalTalk is in use (on the computer this image was taken from). The typical user would see only a General MIDI option, under MIDI synthesizers, and Apple Music under Software Synthesizers. Other choices, such as the ones visible below, would appear if appropriate system extensions have been installed.



APPENDIX

GENERAL MIDI INSTRUMENT NUMBERS

General MIDI Instrument Numbers

1	Acoustic Grand Piano	33	Wood Bass
2	Bright Acoustic Piano	34	Electric Bass Fingered
3	Electric Grand Piano	35	Electric Bass Picked
4	Honky-tonk Piano	36	Fretless Bass
5	Rhodes Piano	37	Slap Bass 1
6	Chorused Piano	38	Slap Bass 2
7	Harpsichord	39	Synth Bass 1
8	Clavinet	40	Synth Bass 2
9	Celesta	41	Violin
10	Glockenspiel	42	Viola
11	Music Box	43	Cello
12	Vibraphone	44	Contrabass
13	Marimba	45	Tremolo Strings
14	Xylophone	46	Pizzicato Strings
15	Tubular bells	47	Orchestral Harp
16	Dulcimer	48	Timpani
17	Draw Organ	49	Acoustic String Ensemble 1
18	Percussive Organ	50	Acoustic String Ensemble 2
19	Rock Organ	51	Synth Strings 1
20	Church Organ	52	Synth Strings 2
21	Reed Organ	53	Aah Choir
22	Accordion	54	Ooh Choir
23	Harmonica	55	Synvox
24	Tango Accordion	56	Orchestra Hit
25	Acoustic Nylon Guitar	57	Trumpet
26	Acoustic Steel Guitar	58	Trombone
27	Electric Jazz Guitar	59	Tuba
28	Electric clean Guitar	60	Muted Trumpet
29	Electric Guitar muted	61	French Horn
30	Overdriven Guitar	62	Brass Section
31	Distortion Guitar	63	Synth Brass 1
32	Guitar Harmonics	64	Synth Brass 2

General MIDI Instrument Numbers (continued)

65	Soprano Sax	97	Ice Rain
66	Alto Sax	98	Soundtracks
67	Tenor Sax	99	Crystal
68	Baritone Sax	100	Atmosphere
69	Oboe	101	Bright
70	English Horn	102	Goblin
71	Bassoon	103	Echoes
72	Clarinet	104	Space
73	Piccolo	105	Sitar
74	Flute	106	Banjo
75	Recorder	107	Shamisen
76	Pan Flute	108	Koto
77	Bottle blow	109	Kalimba
78	Shakuhachi	110	Bagpipe
79	Whistle	111	Fiddle
80	Ocarina	112	Shanai
81	Square Lead	113	Tinkle bell
82	Saw Lead	114	Agogo
83	Calliope	115	Steel Drums
84	Chiffer	116	Woodblock
85	Synth Lead 5	117	Taiko Drum
86	Synth Lead 6	118	Melodic Tom
87	Synth Lead 7	119	Synth Tom
88	Synth Lead 8	120	Reverse Cymbal
89	Synth Pad 1	121	Guitar Fret Noise
90	Synth Pad 2	122	Breath Noise
91	Synth Pad 3	123	Seashore
92	Synth Pad 4	124	Bird Tweet
93	Synth Pad 5	125	Telephone Ring
94	Synth Pad 6	126	Helicopter
95	Synth Pad 7	127	Applause
96	Synth Pad 8	128	Gunshot

GENERAL MIDI DRUMKIT NUMBERS

General MIDI DrumKit Numbers

35	Acoustic Bass Drum	51	Ride Cymbal 1
36	Bass Drum 1	52	Chinese Cymbal
37	Side Stick	53	Ride Bell
38	Acoustic Snare	54	Tambourine
39	Hand Clap	55	Splash Cymbal
40	Electric Snare	56	Cowbell
41	Lo Floor Tom	57	Crash Cymbal 2
42	Closed Hi Hat	58	Vibraslap
43	Hi Floor Tom	59	Ride Cymbal 2
44	Pedal Hi Hat	60	Hi Bongo
45	Lo Tom Tom	61	Low Bongo
46	Open Hi Hat	62	Mute Hi Conga
47	Low -Mid Tom Tom	63	Open Hi Conga
48	Hi Mid Tom Tom	64	Low Conga
49	Crash Cymbal 1	65	Hi Timbale
50	Hi Tom Tom	66	Lo Timbale

GENERAL MIDI KIT NAMES

General MIDI Kit Names

1	Dry Set
9	Room Set
19	Power Set
25	Electronic Set
33	Jazz Set
41	Brush Set
65-112	User Area
128	Default

INDEX

A

AddTrackReference function 16
amplitude 169
asynchronous decompression, scheduled 43
audio level 169

B

beats-per-minute 177
built-in 167, 173

C

cantSendToSynthesizerErr 230, 232, 234, 236, 237, 238, 239, 240, 243, 244, 245, 246, 248, 249, 250, 251, 252, 253, 254, 255, 256, 258, 259, 260, 249, 251, 252
CDBandDecompress function 47
CDCodecFlush function 47
CDCodecSetTimeCode function 48
CDPreDecompress function 46
chunk size, getting preferred 22
chunk size, setting preferred 21
clock 193
CloseComponent Component Manager function 144, 152
CodecCapabilities structure 44
CodecDecompressParams structure 44
Component Interfaces 187
Component Manager 92
  CloseComponent function 144, 152
  component flags value 88, 92
  component subtype value 88, 92
  component type value 88, 92
  FindNextComponent function 88
  manufacturer value 88, 92
  OpenComponent function 87
  OpenDefaultComponent function 87
  selector values for data handler components 93
compressor capability structure 44
configuration information 172, 222, 255, 256
container 86
  assigning 141
  creating 155
  retrieving 142
controller 172, 174, 175, 178, 179, 184, 185, 189, 199, 200, 208, 209, 210, 211, 213, 234, 253, 259, 260, 249, 250, 254
Controller Event 184
Controller events 178
Conversion of Standard MIDI 176
ConvertMovieToFile function 8
copyright 172, 220, 221, 250
cursor, hiding 50
_ControlController 184
_ControlValue 184

D

data handler components 3
  appending data 91, 152
  asynchronous read 145-148
  asynchronous write 153-154
  block size, preferred 156
  buffers, flushing read 158
  buffers, flushing write 159
  cancelling a scheduled read 149
  capabilities, determining 88
  ceding processor time to a handler 158
  closing data reference after read 143
  closing data reference after write 152
  completion function 90, 145-148, 149, 153-154, 160-161
  component flags value 88, 92
  Component Manager 82
  component subtype value 86, 88, 92, 138
  component type value 88, 92
  connection, opening 87
  creating a data handler component 92
  data reference
    types 86
  device index 137
  duties 85-86
  enlarging a data reference 157
  extending a data reference 157
  flushing cached reads 158
  flushing cached writes 159
  free space, getting 157
  hints, playback 159
  index, device 137
  manufacturer value 88, 92
  media handler components 85
  mounting volumes 135
  movie data, reading 90
  movie data, writing 90, 91

networked-device support 134
opening data reference for read 90, 143
opening data reference for write 151
playback hints 159
pre-roll operations 147
priority of read requests 148
processor time, granting to data handler 91, 158
quality of service 89, 132, 134, 148
queued requests, completing 149
QuickTime
versions supported 81
QuickTime for Windows 81-82
version supported 81
random write 153-154
read, asynchronous 90, 145-148
read, synchronous 90, 144
read-ahead time, indicating preferred 150
reading movie data 90
removable volumes 135
responsibilities 85-86
retrieving movie data 90
schedule record 146-148
scheduled read 90, 145-148
scheduled read, cancelling 149
scheduled read, completing 149
selecting 88
selecting a data handler component 86
selecting with Movie Toolbox 22
selector values 93
size, getting data reference 155
size, setting data reference 154
storing movie data 90-91
subtype value, component 86, 88, 92, 138
synchronous read 144
synchronous write 152
type value, component 88, 92
unmounting volumes 135
volume list, getting 89, 132
write, asynchronous 91, 153-154
write, synchronous 91, 152
writing movie data 91
data reference
and component subtype value 92, 138
assigning to a data handler 89, 138
closing after read 143
closing after write 152
comparing 89, 140
creating container for 155
determining ability to support 89, 135
enlarging 157
equivalent 89
free space 157
getting size of 155
opening for read 90, 143
opening for write 90, 151
resolving 140
retrieving from a data handler 139
selecting a handler for 87
setting size of 154
several in one media 2, 20
types 88, 92, 138
working with 138
DataHCanUseDataRef function 89, 135
DataHCloseForRead function 143
DataHCloseForWrite function 152
DataHCompareDataRef function 89, 140
DataHCreateFile function 155
DataHFinishData function 90, 149
DataHFlushCache function 145, 148, 153, 158
DataHFlushData function 159
DataHGetData function 90, 144
DataHGetDataRef function 89, 139
DataHGetDeviceIndex function 137
DataHGetFileSize function 155
DataHGetFreeSpace function 157
DataHGetOSFileRef function 142
DataHGetPreferredBlockSize function 156
DataHGetScheduleAheadTime function 150
DataHGetVolumeList function 89, 132
DataHOpenForRead function 90, 143
DataHOpenForWrite function 90, 151
DataHPlaybackHints function 159
DataHPreextend function 157
DataHPutData function 91, 152
DataHResolveDataRef 140
DataHScheduleData function 90, 145-148
DataHScheduleRecord structure 146-148
DataHSetDataRef function 89, 138
DataHSetFileSize function 154
DataHSetOSFileRef function 141
DataHTask function 91, 158
DataHVolumeListRecord structure 133
DataHWrite function 91, 153-154
decompression parameters structure 44
decompression, scheduled asynchronous 43
decompression, scheduling 37
DecompressSequenceFrameWhen function 37
DeleteTrackReference function 17
dropframe timecode 24, 27

Drum names 175, 241, 249
duration 167, 175, 176, 177, 180, 181, 182, 189, 229

E

End Marker 178, 182, 183, 188
End Marker Event 182
engaged 199, 208, 209
equal tempered notes 170
Event Sequence Format 176
Extended Controller Event 185
Extended Note event 178, 181

F

file *see container*
file reference
  assigning 141
  retrieving 142
Fixed Instrument 173, 174
fixed pitch 181
FlattenMovie function 9
FlattenMovieData function 9
frame time structure 46
frequency 168
function name, longest 67

G

General event 177, 178, 188
General MIDI 171, 173, 199, 254, 255, 256, 257
generic synthesizer 173
GetDataHandler function 22, 87
GetMediaPreferredChunkSize function 22
GetMovieColorTable function 14
GetMovieIndTrackType function 15
GetNextTrackReferenceType function 19
GetTrackLoadSettings function 11
GetTrackReference function 18
GetTrackReferenceCount function 20
_GeneralLength 179
_GeneralSubtype 179

H

hiding the cursor 50
hints, playback 1

I

ICMDecompressComplete function 48
ICMFrameTime structure 46
ICMShieldSequenceCursor function 50
illegalChannelErr 239, 240, 253, 254
illegalControllerErr 214, 260, 250
illegalInstrumentErr 232, 245, 246
illegalKnobErr 214, 236, 237, 238, 251, 252, 257, 258
illegalKnobValueErr 214, 236, 237, 238, 251, 252, 257, 258
illegalNoteChannelErr 202, 203, 206, 207, 208, 209, 211, 212, 214
illegalPartErr 234, 242, 243, 244, 245, 246, 250, 253, 254, 255, 256, 258, 259, 260, 249
illegalVoiceAllocationErr 254
Image Compression Manager
  decompression, scheduled asynchronous 43
  decompression, scheduling 37
  timecode information, setting 41
  timecode support 37
image compressor components
  scheduled asynchronous decompression 43-47
  timecode information, setting 48
  timecode support 43
Instrument 167, 173, 174, 175, 181, 185, 215, 216, 217, 220, 231, 236, 237, 241, 242, 243, 245, 246, 247, 249, 255, 256
Instrument Control 241
Instrument index number 178, 184, 186
Instrument number 175, 180, 184, 185, 186, 215, 231, 232, 233, 241, 243, 245, 246, 249
InstrumentAboutInfo 250
InstrumentData 242
_Instrument 180, 184

K

kControllerAfterTouch 213
kControllerBalance 213
kControllerBreath 213
kControllerCeleste 213
kControllerChorus 213
kControllerExpression 213
kControllerFoot 213
kControllerModulationWheel 184, 213
kControllerPan 184, 213
kControllerPhaser 213
kControllerPitchBend 184, 213
kControllerPortamento 213
kControllerPortamentoTime 213
kControllerReverb 213

kControllerSoftPedal 213
kControllerSostenuto 213
kControllerSustain 213
kControllerTremolo 213
kControllerVolume 184, 213
kDataHCanRead flag 133, 135
kDataHCanStreamingWrite flag 134, 137
kDataHCanWrite flag 133, 136
kDataHMustCheckDataRef flag 134
kDataHSpecialRead flag 133, 136
kDataHSpecialReadFile flag 133, 136
kDataHSpecialWrite flag 134, 136
kInstrumentMatchGMNumber 231
kInstrumentMatchName 231
kInstrumentMatchNumber 231
kInstrumentMatchSynthesizerName 231
kInstrumentMatchSynthesizerType 231
Knob 167, 168, 172, 173, 174, 175, 178, 186, 200, 229, 236, 237, 238, 241, 251, 252, 253, 257, 258
Knob Event 186
KnobDescription 251
kTuneDontClipNotes 189
kTuneExcludeEdgeNotes 189
kTuneLoopUntil 190
kTuneQuickStart 189
kTuneStartNow 189
_KnobKnob 186
_KnobValue 186

M

master reference timer 175, 251
media with several data references 2, 20
MediaForceUpdate function 79
MediaGetDrawingRgn function 78
MediaIdle function 77
microtonal 170, 178, 180, 181, 212, 234
Microtones 167
middle C 180, 181, 212
MIDI 164, 165, 167, 170, 171, 172, 173, 175, 176, 199, 204, 205, 207, 212, 222, 223, 224, 225, 226, 227, 229, 231, 232, 234, 235, 236, 237, 239, 240, 243, 244, 248, 250, 251, 252, 253, 254, 255, 256, 258, 259, 260, 249, 251, 252, 253, 254, 255, 256, 257
midiManagerAbsentErr 204, 223
Modifiable Instruments 173
movie data import components
  file type, getting 76
Movie Toolbox
  and data handler components 86, 87
  and removable volumes 135
  color table, getting 14
  color table, setting 13
  data handler, selecting 22
  data references, multiple 2, 20
  drawing-complete function, assigning 12
  forcing it to check your data handler's capabilities 134
  GetDataHandler function 87
  hints 1
  MoviesTask function 158
  preloading tracks 1, 9, 11
  read-ahead time 150
  reads before opening data reference 143, 145, 148
  track references 2, 16
  tracking data handler components 132
  tracks, adding track references 16
  tracks, counting track references 20
  tracks, deleting track references 17
  tracks, modifying track references 18
  tracks, reading track references 18
  tracks, scanning track reference types 19
  tracks, searching by characteristic 15
MovieImportGetFileType function 76
Music Component 164, 165, 167, 168, 170, 171, 172, 173, 174, 175, 187, 199, 203, 210, 222, 225, 229, 230, 241, 247
Music Component Interface 229
Music Media Handler 165
Music Preferences 172, 222, 228
music track 164, 168, 170, 176, 177
musical note 168
MusicFindTone 231
MusicGetController 259
MusicGetDescription 230
MusicGetDrumKnobDescription 252
MusicGetDrumNames 249
MusicGetInstrument 242
MusicGetInstrumentAboutInfo 250
MusicGetInstrumentKnobDescription 251
MusicGetInstrumentNames 247
MusicGetInstrumentNumber 243
MusicGetKnob 236
MusicGetKnobDescription 238
MusicGetMasterTune 251
MusicGetMIDIProc 239
MusicGetPart 253
MusicGetPartKnob 256

MusicGetPartName 255
MusicPlayNote 234
MusicResetPart 259
MusicSetController 249
MusicSetInstrument 243
MusicSetInstrumentNumber 245
MusicSetKnob 236
MusicSetMasterTune 251
MusicSetMIDIProc 240
MusicSetPart 254
MusicSetPartKnob 258
MusicSetPartName 255
MusicStoreInstrument 246
_MarkerSubtype 183
_MarkerValue 183

N

NACopyrightDialog 221
NADisengageNoteChannel 209
NADisposeNoteChannel 202
NAEngageNoteChannel 208
NAFindNoteChannelTone 215
NAGetDefaultMIDIInput 226
NAGetNoteChannelInfo 203
NAGetRegisteredMusicDevice 225
NALoseDefaultMIDIInput 205
NANewNoteChannel 200
NAPickArrangement 220
NAPickInstrument 217
NAPlayNote 212
NAPrerollNoteChannel 206
NARegisterMusicDevice 222
NAResetNoteChannel 210
NASaveMusicConfiguration 228
NASetController 213
NASetDefaultMIDIInput 227
NASetKnob 214
NASetNoteChannelInstrument 216
NASetNoteChannelVolume 211
NAStuffToneDescription 219
NAUnregisterMusicDevice 224
NAUnrollNoteChannel 207
NAUseDefaultMIDIInput 204
new movie, creating from user function 4
NewMovieFromFile function 7
NewMovieFromUserProc function 4
Note 180
Note Allocator 164, 165, 171, 172, 173, 187, 199, 200, 202, 203, 204, 205, 206, 207, 208, 209, 210, 211, 212, 213, 214, 215, 216, 220, 221, 222, 224, 225, 226, 227, 228, 232
Note Channel 168, 173, 199, 200
Note Channel Allocation and Use 199
note channels 171, 196, 197, 199, 207
Note event 176, 177, 178, 179, 182
NoteAllocator errors 188
noteChannelNotAllocatedErr 188, 206
NoteRequest 200
notes 164, 165, 167, 168, 170, 171, 172, 176, 181, 187, 189, 195, 199, 200, 208, 209, 210, 234, 239
_NoteDuration 180
_NotePitch 180
_NoteVelocity 180
_NoteVolume 180

O

OpenComponent Component Manager function 87
OpenDefaultComponent Component Manager function 87
oscillators 169
outputs, sequence grabber 51, 59-67

P

Part 168, 173, 174, 175, 178, 180, 184, 185, 186, 188, 210, 230, 234, 241, 244, 245, 246, 253, 254, 259, 249
Part Access 253
Part number 179, 181, 229, 234, 242, 243, 244, 245, 246, 250, 253, 254, 255, 256, 258, 259, 260, 249
pitch 167, 168, 175, 177, 178, 180, 181, 184, 199, 212, 229, 234, 249
playback hints 1
Polyphony 168, 200, 220
pre-roll operations 90
preloading tracks 1, 9, 11

Q

QMA 164, 165, 167, 168, 169, 170, 173, 176, 178, 188, 239
queued-up 171
queueFlags 190
QuickTime for Windows 90, 92
QuickTime movie track 168, 176
QuickTime Music Architecture 164, 167, 171, 187
QuickTime Music tracks 165

R

real time 193
Rest event 176, 182

rests 164, 167, 176, 178, 187
reverb 174
_RestDuration 182

S

scale 167, 168, 170, 176, 177, 179, 180, 181, 182, 187, 194
sequence 164, 165, 167, 168, 171, 172, 176, 178, 182, 187, 188, 189, 190, 191, 192, 193, 194, 195, 196, 198, 199
Sequence Control 191
sequence data 164, 188
sequence grabber channel components
  maximum data rate, getting 70
  maximum data rate, setting 69
sequence grabber component
  output, assigning to a channel 63
sequence grabber components
  destination, determining 54
  destination, specifying 51
  mode, determining 57
  output, configuring 64
  output, creating a new 59
  output, disposing of 62
  output, getting remaining space 67
  outputs 51, 59-67
  timecode source identification information, getting 58
  timecode source identification information, setting 58
  timecode support 51
sequence grabber outputs 51, 59-67
SetDSequenceTimeCode function 41
SetMediaDefaultDataRefIndex function 20
SetMediaPreferredChunkSize function 21
SetMovieColorTable function 13
SetMovieDrawingCompleteProc function 12
SetTrackLoadSettings function 9
SetTrackReference function 18
SGChannelGetDataSourceName function 58
SGChannelGetRequestedDataRate function 70
SGChannelSetDataSourceName function 58
SGChannelSetRequestedDataRate function 69
SGDisposeOutput function 62
SGGetDataOutputStorageSpaceRemaining function 67
SGGetDataRef function 54
SGGetMode function 57
SGNewOutput function 59
SGSetChannelOutput function 63
SGSetDataRef function 51
SGSetOutputFlags function 64
shielding the cursor 50
slaved 193
slots 173, 198
SMPTE timecode information 24
software component 164, 165, 167, 168, 171, 173
software synthesizer 173
standard note 170, 178, 179, 180, 181
stopFlags 191
subtype 178, 179, 182, 183
synthesizer 164, 165, 167, 168, 169, 170, 171, 172, 173, 174, 175, 178, 179, 187, 199, 200, 207, 212, 222, 225, 229, 230, 231, 232, 234, 235, 236, 237, 238, 239, 240, 242, 243, 244, 245, 246, 248, 249, 250, 251, 252, 253, 254, 255, 256, 258, 259, 260, 249, 251, 252, 254
Synthesizer Access 229
Synthesizer Timing 251
SynthesizerConnections 222, 226
SynthesizerDescription 230
SynthesizerErr 223, 230, 232, 234, 236, 237, 238, 239, 240, 242, 243, 244, 245, 246, 247, 249, 250, 251, 252, 253, 254, 255, 256, 258, 259, 260, 249, 251
System Configuration 222
_StuffControlEvent 184
_StuffGeneralEvent 179
_StuffKnobEvent 186
_StuffNoteEvent 180
_StuffRestEvent 182
_StuffXControlEvent 185
_StuffXNoteEvent 181

T

TCFrameNumberToTimeCode function 31
TCGetCurrentTimeCode function 29
TCGetDisplayOptions function 35
TCGetSourceRef function 33
TCGetTimeCodeAtTime function 29
TCGetTimeCodeFlags function 34
TCSetDisplayOptions function 35

TCSetSourceRef function 32
TCSetTimeCodeFlags function 33
TCTimeCodeToFrameNumber function 30
TCTimeCodeToString function 32
timbrality 168
Timbre 168
timbres 170
time scale 176, 180, 182, 187, 194
TimeBase 193
timecode definition structure 27-28
timecode media handler 3, 23-36
  adding samples 26
  and track references 25
  control flags, getting 34
  control flags, setting 33
  converting frame number to timecode time 31
  converting timecode time to a string 32
  converting timecode time to frame number 30
  converting timecode to media time 29
  creating timecode media 25
  display options, getting 35
  display options, setting 35
  displaying timecode information 25, 35
  dropframe technique 24, 27
  getting timecode information 29
  sample description 26
  source identification information 25
  source identification information, getting 33
  source identification information, setting 32
  timecode definition structure 27
  timecode record 28
timecode media, creating 25
timecode record 28
timing 164, 165, 168, 171, 172, 174, 175, 187, 193
ToneDescription 215, 242
ToneDescription { 200
track 167, 168, 220
  adding track reference 16
  counting track references 20
  deleting track reference 17
  modifying track reference 18
  preloading 1, 9, 11
  reading track reference 18
  reference 2, 16
  scanning track reference types 19
  searching by characteristic 15
track references 2, 16
  used with timecode media 25
Tune Player 164, 165, 168, 171, 172, 173, 176, 177, 178, 182, 187, 188, 189, 191, 192, 193, 194, 195, 196, 197, 198
TuneGetIndexedNoteChannel 197
TuneGetStatus 198
TuneGetTimeBase 193
TuneGetTimeScale 194
TuneGetVolume 191
TuneInstant 195
tuneParseErr 188
tunePlayerFullErr 190
TunePreroll 196
TuneQueue 189
TuneSetHeader 178, 180, 181, 184, 185, 186, 187, 188, 197
TuneSetTimeScale 194
TuneSetVolume 192
tuneStartPosition 190
TuneStop 191
tuneStopPosition 190
TuneUnroll 196

U

units of time 180, 182
units-per-second 167, 194
user interface dialogs 172, 220
user-modifiable Instruments. See Modifiable Instruments 173

V

VDGetTimeCode function 73
VDSetDataRate function 71
velocity 167, 177, 180, 199, 212, 234
video digitizer components
  limiting data rate 71
  timecode information, retrieving 73
  timecode support 71
Voice 169
volume 167, 168, 169, 172, 175, 180, 181, 187, 191, 192, 199, 211, 212, 229, 234, 259
Windows, QuickTime support *see QuickTime for Windows*

X

_XControlController 185
_XControlValue 185
_XInstrument 179, 182, 185, 186
_XNoteDuration 182
_XNotePitch 182

_XNoteVelocity 182
_XNoteVolume 182