# Contents

## QuickTime for Windows Developer's Manual



## Section I. QuickTime for Windows Overview



## Section 2. A QuickTime for Windows Tutorial



## Section 3. Programmer's Reference



## Appendices

# Preface

## About the QuickTime for Windows Documentation

This document is the official programmer's manual for developers of QuickTime for Windows-aware applications in the Microsoft Windows environment. Unlike the Macintosh version, the current release of QuickTime for Windows handles movies in play mode only. As a result, this manual focuses on an QuickTime for Windows entity known as the Movie Controller. All movies must be under the direct supervision of movie controllers, and most of the programmatic interface presented in the *Tutorial* and *Programmer's Reference* sections of this on-line manual is devoted to supporting the creation and functionality of this entity.

This approach was taken because much of the existing documentation for QuickTime for Windows on the Macintosh covers implementation areas not yet available to the Windows developer. General architectural overviews and design perspectives of QuickTime for Windows *are* covered, but material which could distract or otherwise prevent developers from running movies in their Windows programs as soon as possible has been kept to a minimum. If the developer wishes further information on how movies are created and edited, or about the internals of QuickTime for Windows itself, he or she should consult the Apple QuickTime documentation.

To get the greatest benefit from this manual, the developer should already be familiar with the Windows development tools and the Microsoft C programming language environment.

# A. QuickTime for Windows Concepts

As a QuickTime for Windows developer, you will need to understand the various high level strategies and paradigms that QuickTime for Windows incorporates before you design and code your own QuickTime for Windows applications. These concepts fall into several categories: what QuickTime for Windows is, how programs incorporate it, what is normal QuickTime for Windows behavior and what is the responsibility of the application, and so forth. This section gives you enough background on these concepts to proceed to the tutorial section and start writing your own QuickTime for Windows programs.

## Related Topics:

# 1. What is QuickTime for Windows?

QuickTime for Windows is a technology that lets your Microsoft Windows programs play QuickTime movies and view QuickTime pictures. QuickTime is Macintosh-based software that can create movies as well as play them.

A movie playing in a Windows application can be directly manipulated by the user with a special control bar called a movie controller, usually found attached to the bottom of the movie window. Any Windows program can play one or more QuickTime for Windows movies, from sophisticated word processors and spreadsheets to standalone applications created specifically to play movies.

To make your Windows programs QuickTime for Windows-capable, you will have to modify their source code, recompile and relink them with the QuickTime for Windows libraries. This document will guide you through that process.

## 2. Movies and Time

A traditional movie, whether stored on film, laser disk or tape, is a continuous stream of data. To the Windows developer, a QuickTime movie is a standard DOS file with an extension of .MOV. A movie file contains digitized visual and sound data along with sequencing information describing the order in which the movie frames should be played. When the file is opened, the data is assigned a *movie object*. It is still not playable as a movie, however, until it is associated with a movie controller.

Movies may be played on Windows machines, but not saved in any form. You must use Macintosh-based QuickTime software to edit movies. An individual movie frame may be copied to the Windows clipboard. Of course, movie files can be copied or renamed outside of QuickTime for Windows applications just like any other DOS files. Further information on Macintosh QuickTime movie files can be found in the QuickTime documentation.

A QuickTime movie is completely self-contained. All of its visual and sound data exists in a single DOS file, which is referenced by a QuickTime for Windows program through QuickTime for Windows API calls when the time comes to load it. Your application need never work directly with movie data, as QuickTime for Windows routines allow your programs to manage movie content and characteristics while they are playing under Windows.

Movies are instantiated and later freed by several QuickTime for Windows functions. OpenMovieFile opens the file containing the movie, just like any DOS file. NewMovieFromFile extracts movie data from the opened file and assigns a movie object to that data. This object is the means by which the movie will be played. CloseMovieFile closes the file normally. DisposeMovie frees the movie object.

```
MovieFile mfMovie;

Movie mMovie;

•

•

OpenMovieFile ("MYMOVIE.MOV", &mfMovie, OF_READ);

NewMovieFromFile (&mMovie, mfMovie, NULL, NULL, 0, NULL);

CloseMovieFile (mfMovie);

•

•

DisposeMovie (mMovie);
```

Understanding time management of media is essential to understanding QuickTime for Windows routines and data structures. QuickTime for Windows defines *time coordinate systems* that anchor a movie to a common temporal reality--the second. A time coordinate system contains a time scale scored in time units. The number of units that pass per second quantifies the scale. For example, a time scale of 26 means that 26 units pass per second and each time unit is 1/26 of a second.

A time coordinate system also contains a duration, which is the length of the movie in number of time units it contains. Particular points in a movie can be identified by a time value, which is the number of time units to that point.

The last of QuickTime for Windows time-related concepts is the idea of rate. A movie's rate is expressed as a multiple of its time scale. For instance, in a movie with a time scale of 2 played at rate of 2.5, five time units would pass in one second.

## 3. Active and Inactive Movies

Movies have active and inactive states. The most distinctive feature of an inactive movie is that it simply cannot be played. QuickTime for Windows accomplishes this by not giving the movie any time slices from its internal scheduler. Visually, the movie appears to be paused, but any attempt to start it will fail until the movie is activated.

You can make a movie active or inactive when you extract it from a file, or change its state later. In the code fragment above, the movie is made inactive by setting the fifth parameter of NewMovieFromFile to `0`. Using `newMovieActive` instead makes it active:

```
MovieFile mfMovie;

Movie mMovie;

•

•

NewMovieFromFile (&mMovie, mfMovie, NULL, NULL, newMovieActive, NULL);
```

To set the movie's state dynamically, you can use the routine SetMovieActive:

```
Movie mMovie;

BOOL bState;

•

•

SetMovieActive (mMovie, bState);
```

A movie's state can be queried via the function GetMovieActive.

It is good QuickTime for Windows style to keep a movie inactive until you are ready to play it, since active movies receive cycles from QuickTime for Windows' scheduler and are a drag on the system unless ready for play. You should therefore use normally `0` instead of `newMovieActive` when calling NewMovieFromFile, and subsequently SetMovieActive once you are ready to play the movie.

# 4. The Movie Controller

As noted above, the user interface to a QuickTime for Windows movie is the Movie Controller. Any movie played in a Windows application must be associated with one. Normally, a movie controller appears as a bar-shaped collection of controls attached to the bottom edge of a movie. Each of the individual elements in a movie controller dictates a specific action for a movie:

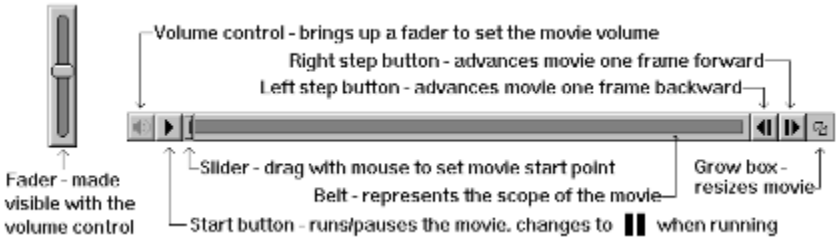



Figure 1. The Movie Controller

Under certain circumstances, some movie controller elements may not be present. For example, your application might need to restrict the operation of a controller by not displaying the step buttons. Or, the user could use the grow box to shrink it to the point where the controller itself must hide some of its elements, based on the available space it has to work with. A movie controller instance is created and later freed with the routines NewMovieController and DisposeMovieController:

```
RECT rcMovie;
Movie mMovie;
MovieController mcController;
•
•
mcController = NewMovieController (mMovie, &rcMovie, mcTopLeftMovie,
   hwndParent);
•
•
DisposeMovieController (mcController);
```

In Windows terms, a movie and its associated controller have a common parent window, generally the application in whose client area they both appear. When adding movie controllers to your applications, you can think of them as custom controls that are subject to the same conventions and programmatic considerations as standard Windows controls.

You should note that while destroying a window that contains a movie controller causes DisposeMovieController to be called internally, this is simply a safety feature. You should dispose your movie controllers explicitly as a matter of course.

Although the Movie Controller is clearly designed to accept mouse input, it has a keyboard interface as well. The following table applies to any movie controller with an enabled keyboard interface:

| Key | Action |
|---|---|
| F1 | Brings up the Help subsystem (not part this of interface per se--actually the responsibility of the application) |
| Return/Space | Toggles Play/Pause state |
| Right Arrow | Step forward one frame |

| | |
|---|---|
| Left Arrow | Step backward one frame |
| Up Arrow | Increase volume (when sound is enabled) |
| Down Arrow | Decrease volume (when sound is enabled) |
| Home | Go to start of movie |
| End | Go to end of movie |
| Ctrl + Home | Go back to next interesting time* |
| Ctrl + End | Go forward to next interesting time* |
| Ctrl + Right Arrow | Play forward |
| Ctrl + Left Arrow | Play backward |
| Shift + (Return or Space) | Plays and selects while playing, until shift is released |
| Shift + Right Arrow | Extends selection criteria through the next frame |
| Shift + Left Arrow | Extends selection criteria through the previous frame |
| Shift + Home | Go to start of movie, extending selection back to start |
| Shift + End | Go to end of movie, extending selection to end |
| Ctrl + Shift + Home | Go back to next interesting time, extending selection* |
| Ctrl + Shift + End | Go forward to next interesting time, extending selection* |

**Interesting times* are normally the start and end points of movies and selections (if any).

The focus of this manual will be the Movie Controller. The API is rich enough, however, to allow movies to be handled in a wide variety ways to make your QuickTime for Windows programs robust and interesting to use.

## 5. Initialization and Termination of QuickTime for Windows Programs

Initializing your applications to play movies is essentially a three-step process. First, links to QuickTime for Windows must be established. Second, you have to allocate QuickTime for Windows memory for your application. Finally, you must add a routine to your application's main window procedure.

```
OSErr oserrResult;
•
•
if ((oserrResult = QTInitialize (NULL)) != QT_OK)
   {
   /* Take appropriate action, e.g. a message box saying movies won't */
   /* play but the program will continue to run. */
   }
if (EnterMovies () != noErr)
   {
   /* Take appropriate action, e.g. a message box saying movies won't */
   /* play but the program will continue to run. */
   }
```

Establishing links to QuickTime for Windows is accomplished by calling the routine QTInitialize. Normally, this is done automatically when the first QuickTime for Windows call is executed, but it is good style to call it yourself. This function takes one parameter, the address of a variable which is filled with QuickTime for Windows version data that might be useful if your application depends on it. If no error condition is returned, you must call EnterMovies to allocate QuickTime for Windows memory for your application. If either QTInitialize or EnterMovies returns an error, such as incorrect Windows version or sub-386 CPU, your application will run normally but all subsequent movie-related calls will be ignored by QuickTime for Windows.

It is only necessary to call QTInitialize once in each of your applications. If a particular application employs DLLs that make QuickTime for Windows API calls, each DLL can initialize itself by calling QTInitialize explicitly. This is recommended as good QuickTime for Windows style and can be done in LibMain:

```
int FAR PASCAL LibMain (HINSTANCE hInst, WORD wDataSeg,
   WORD wHeapSize, LPSTR lpszCmdLine)
   {
   OSErr oserrResult;
   •
   •
   if ((oserrResult = QTInitialize (NULL)) != QT_OK)
      {
      /* Take appropriate action, e.g. a message box saying movies */
      /* won't play but the program will continue to run. */
      }
   •
   •
   return 1;
   }
```

Calling EnterMovies is necessary to play movies (your program might display just QuickTime for Windows pictures, in which case the only initialization required is QTInitialize). EnterMovies only needs to be called once by your program (or its DLLs) to initialize it for playing movies--subsequent calls to EnterMovies are ignored by QuickTime for Windows.

The final piece of code required to make movies run is MCIsPlayerMessage, a function that must be placed in the application's window procedure. For each movie controller that your program creates, there must be a separate call to this routine in the movie controller's parent window procedure.

MCIsPlayerMessage processes all messages coming into the window procedure, but only messages directed to its associated controller receive attention. Movies are started and stopped, their states and attributes changes, etc., based on messages being routed to their controllers via this routine.

```
LONG FAR PASCAL WndProc (HWND hWnd, UINT uiMessage, WPARAM wParam,
   LPARAM lParam)
   {
   if (MCIsPlayerMessage (mcController, hWnd, uiMessage, wParam,
      lParam))
      return 0;
   switch (uiMessage)
      {
      /* cases */
      }
   return DefWindowProc (hWnd, uiMessage, wParam, lParam);
   }
```

Now that we have established the paradigm for what keeps movies running, we can make an exception to it. You don't always have to use MCIsPlayerMessage, especially if your program functions in an unusual way. There are essentially two QuickTime for Windows API calls that handle movie playing in this case: MCIdle and MCKey. You can refer to the information on these routines to see how they work. If your program can accommodate MCIsPlayerMessage, however, it is highly recommended that you code it that way.

At this point, your application as a whole is considered initialized under QuickTime for Windows, even though no movies or movie controllers have yet been instantiated.

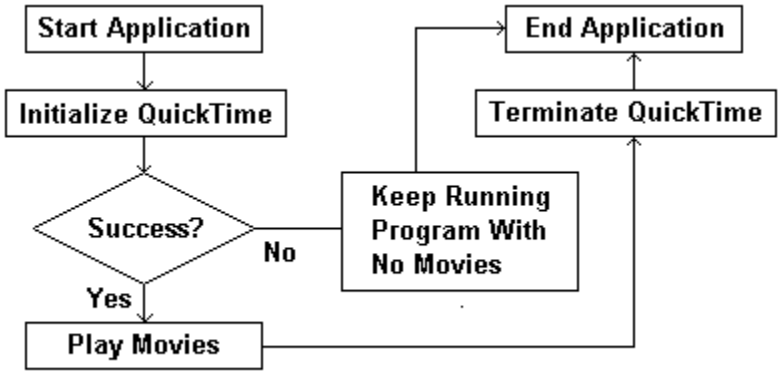



Figure 2. Initialization and Termination

Graceful termination of QuickTime for Windows programs that play movies is almost a mirror image of initialization. At some point in your program's termination activity, the routines that deallocate QuickTime for Windows memory and sever links to the QuickTime for Windows libraries must be called.

•

```
// Deallocate QuickTime for Windows memory
   ExitMovies ();
// Sever links to QuickTime for Windows
   QTTerminate ();
```

Although QTTerminate is called automatically when your program or DLL terminates, it is still good style to issue the call explicitly. In some cases, you may want to call it way before the normal end of your application (e.g., when system memory is at a premium and your program is finished playing movies).

If your program uses DLLs with QuickTime for Windows routines, each DLL can call QTTerminate. This is the recommended approach and can be done in the `WEP` function:

```
int FAR PASCAL WEP (int nParam)
   {
   •
   •
// Sever links to QuickTime for Windows
   QTTerminate ();
   return 1;
   }
```

QuickTime for Windows programs that do not call EnterMovies (e.g. those that display only individual QuickTime for Windows pictures) do not have to call ExitMovies. Like EnterMovies, you only need to call ExitMovies once during the life of your program.

## 6. Associating Movies with Movie Controllers

As noted earlier, a movie must be associated with a controller before it can be played. Several routines in the QuickTime for Windows API perform this operation. For an initial association, NewMovieController is commonly used, as we saw earlier.

For existing controllers, a good choice is MCNewAttachedController. You need to supply parameters for the existing movie and movie controller objects, the window handle of the parent application and the upper left corner of the movie rectangle.

```
Movie mMovie;
MovieController mcController;
POINT ptUpperLeft;
•
•
MCNewAttachedController (mcController, mMovie, hWnd, ptUpperLeft);
```

MCSetMovie takes the same parameters and lets you set the movie object to NULL (second parameter) if you want to specifically disassociate the controller from the movie.

```
Movie mMovie;
MovieController mcController;
POINT ptUpperLeft;
•
•
MCSetMovie (mcController, mMovie, hWnd, ptUpperLeft);
```

When a controller is associated with a movie, the movie object reference is recorded in the controller's data structure. A movie controller can be associated with many movies during its existence, but only one at a time (see figure 4, below). Movie data structures contain no elements which link them with movie controllers.

Once a movie is associated with a controller, it starts playing immediately (assuming it has a non-zero play rate, which is normally the case). To make a movie paused when first visible and associated with a new controller, you can use MCDoAction with an action of mcActionPlay and a play rate of 0. It is good style to do this as soon as possible after performing the association.

```
Movie mMovie;
MovieController mcController;
RECT rcMovie;
•
•
mcController = NewMovieController (mMovie, &rcMovie,
   mcTopLeftMovie + mcScaleMovieToFit, hWnd);
MCDoAction (mcController, mcActionPlay, 0);
```

If you want to play *n* cases of the same movie simultaneously, you have to open the file *n* times to get *n* unique movie objects, then associate *n* controllers.

Movie controllers remain associated with movies regardless of their states. If a controller is made invisible or inactive, for instance, it stays associated with its movie. Conversely, movies continue to play even if the

states of their associated controllers are changed while they are playing. If either one of an associated pair is destroyed, the other is not affected.

Association implies nothing about the proximity of movies and their controllers on the screen. It is simply the means by which any movie can be plugged in to any controller and played.

# 7. Playing Movies through a Movie Controller

A movie associated with a controller is ready for playing (if the movie is active). While the basic apparatus for this activity appears simple and straightforward, there are many subtleties in the relationship of the movie controller to the movie. In one sense, the Movie Controller is simply a human interface. In another, it is the mechanism through which large amounts movie data are focused and made meaningful to the user.

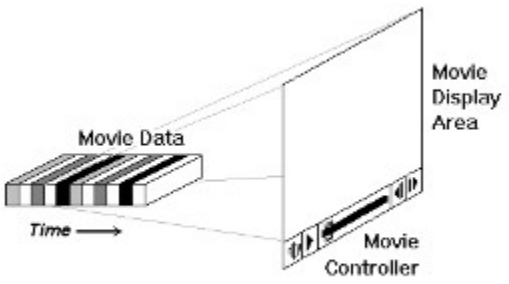



Figure 3. Relationship of Movie Controller to Movie Data

Individual elements of the controller calibrate this mechanism by determining movie sound volume, movie start point, movie display size, etc. Most of these elements change their appearance depending on the values they represent. One element, the volume fader, does not appear at all until specifically called up.

An important distinction needs to be made here: The visual representation of a movie is the sequence of images which flow through a rectangular area on your screen, even though the movie is actually the chunk of movie *data* sitting in memory. It is the Movie Controller, acting as a movie projector, that is the connection between the movie data and its presentation (i.e. it tells the movie to start and stop playing but also specifies the attributes of the area in which the movie will appear).

A movie is started by the function MCDoAction with the mcActionPlay action parameter and an appropriate play rate. This can happen internally when a movie controller's play button is clicked, or overtly at any appropriate place in your program.

```
Movie mMovie;
MovieController mcController;
LFIXED lfxRate;
•
•
lfxRate = GetMoviePreferredRate (mMovie);
MCDoAction (mcController, mcActionPlay, lfxRate);
```

As a movie plays, a synchronized stream of data in the form of still image frames is sent to the specified movie display area according to the settings held by the movie controller. Similarly, blocks of movie sound data are sent to your system's sound driver after being synchronized with the visual data.

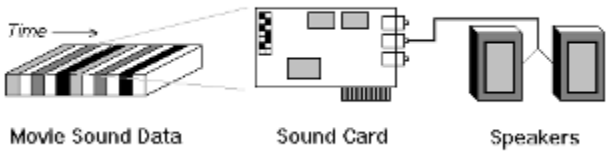



Figure 4. Movie Sound Data Handling

## 8. Attached and Detached Movie Controllers

Until now, we have only been concerned with one type of movie controller--the attached variety. A controller's underlying autonomy, however, is demonstrated by the fact that it can be visually detached from a movie and still play it. Detached controllers can be repositioned anywhere on the screen and still remain associated with their movies, just as if they were still physically attached. They may be disabled, hidden and resized in their detached state as well.

Detachment is a two-step process if you want the controller visually separated from the movie. The most commonly used routines are MCSetControllerAttached with its last parameter set `FALSE` (resets the attachment flags) and MCPositionController (specifies new coordinates):

```
MovieController mcController;
RECT rcMovie, rcController;
•
•
MCSetControllerAttached (mcController, FALSE);
MCPositionController (mcController, &rcMovie, &rcController, 0L);
```

Once detached, a movie controller can be easily re-attached via another call to the function MCSetControllerAttached, this time with `TRUE` as the last parameter. The controller will move back to its normal attached position beneath the movie it controls.

You can query the attachment state of a controller using `MCIsControllerAttached` and also resize it independently from its movie after it has been detached. A detached controller cannot resize its associated movie.

*Note: A detached controller cannot be in a different window than that of its movie.*

---

Although attached movie controllers are the most straightforward way to direct the operation of your movies, it is easy to conceive of interesting ways to use detached controllers. For instance, they could have specific meanings or implications in a customized user interface, or they could control movies which have been built into other graphical objects without getting in the way. Detachment can be viewed as simply an attribute of an associated movie/movie controller pair.

## 9. Active and Inactive Movie Controllers

Instantiated movie controllers exist in one of two states as far as QuickTime for Windows is concerned: active or inactive. When a controller is created, it is set to the active state by default. At any point in the program, it may be set to the inactive state by calling MCActivate with its last parameter set to `FALSE`. Calling the function with `TRUE` reactivates the movie controller.

```
MovieController mcController;
```

- 
- 

```
MCActivate (mcController, hWndParent, FALSE);
```

Generally, movie controllers behave very much like standard Windows controls. An inactive movie controller is especially analogous to a disabled Windows control in that it does not respond to mouse clicks. Additionally, all of its elements are grayed, the slider appears as an outline and the belt is hidden. Keyboard input is enabled/disabled separately.

QuickTime for Windows allows you to set the active or inactive state for as many movie controllers as you wish. If one of your applications requires that only a single controller have active status at any given time, you will have to devise your own scheme for managing these types of situations.

Both attached and detached movie controllers can be made inactive. Doing so has no effect on the movie with which either type is associated, except that the movie cannot be affected by the controller user interface until it is reactivated.

If a movie is running and its controller is inactive, you either have to call a function like MCDoAction with appropriate parameters or reactivate the controller to allow the user to stop the movie. There is no QuickTime for Windows function to specifically query the active state of a movie controller.

The ability to alter the state of a movie controller dynamically could be advantageous under a number of scenarios. For instance, you might have a movie that your application needs to play uninterrupted from beginning to end. In this instance, you would disable the controller when the movie was started and re-enable it when the movie was over.

Another example is the case mentioned earlier where you want only one of many movie controllers active at a time, so that keyboard input can be directed properly. As your QuickTime for Windows applications increase in complexity, this level of control will prove valuable.

## 10. Movie Size and Position

### *Bounds Rectangles*

The key to sizing and positioning movies and movie controllers is the controller's *bounds rectangle*. If the movie controller is attached, this is the area encompassed by the controller plus the movie rectangle. When a movie controller is detached, its dimensions alone determine the bounds rectangle. Rectangles specified by routines which move or create movie controllers become the bounds rectangles for those controllers. Depending on the particular function (and possibly its flags), the resulting bounds rectangle treats its contents in different ways. In some cases, the movie is scaled within the limits of the bounds rectangle. In others, the movie is resized to completely fill its assigned portion of the rectangle. After any call that resizes or repositions the bounds rectangle is processed, QuickTime for Windows calls MCDoAction with mcActionControllerSizeChanged. If your program has a filter, you can make it handle this action (see the section on filters for further information).

### *NewMovieController*

This call creates a new attached controller in the bounds rectangle you provide. The movie and controller are positioned in the rectangle according to the creation flags specified. The following example shows how a new movie controller is created with a bounds rectangle matching the natural dimensions of a movie plus the controller, then how the dimensions of the bounds rectangle are retrieved so that the movie/movie controller pair can be exactly encompassed by the parent window:

```
MovieController mcController;
Movie mMovie;
RECT rcMovie;
•
•
// Get natural dimensions of movie
   GetMovieBox (mMovie, &rcMovie);
   OffsetRect(&rcMovie, -rcMovie.left, -rcMovie.top);
// Instantiate the controller
   mcController = NewMovieController (mMovie, &rcMovie,
      mcTopLeftMovie + mcScaleMovieToFit, hWnd);
// Get the new bounds rectangle
   MCGetControllerBoundsRect (mcController, &rcMovie);
   AdjustWindowRect (&rcMovie, WS_CAPTION | WS_OVERLAPPED, FALSE);
   OffsetRect(&rcMovie, -rcMovie.left, -rcMovie.top);
   SetWindowPos (hWnd, 0, 0, 0,
      rcMovie.right, rcMovie.bottom, SWP_NOMOVE | SWP_NOZORDER);
   ShowWindow (hWnd, nCmdShow);
   UpdateWindow (hWnd);
```

### *MCSetControllerBoundsRect*

For detached movie controllers, MCSetControllerBoundsRect repositions and resizes the controller. For attached controllers, it repositions and resizes both the controller and the movie.

### *MCPositionController*

This routine repositions the movie and movie controller for both attached and detached controllers:

**Detached Controllers:** Calling MCPositionController for a detached controller requires specifying two rectangles, one for the movie and one for the controller. The controller is always centered vertically in its rectangle. The function returns `controllerBoundsNotExact` if this rectangle is too big. The movie is repositioned and resized depending on the flags you provide.

**Attached Controllers:** Calling MCPositionController on an attached controller requires specifying only one rectangle for both the movie and the controller (the second rectangle is ignored and should be coded as NULL). The way the rectangle is used depends on the flags you provide.

### *MCSetControllerAttached*

As discussed previously, MCSetControllerAttached attaches or detaches a movie controller. If the controller is made detached, only a logical operation takes place. It is not physically moved until a subsequent MCPositionController is issued. If the movie controller is made attached, it is moved underneath the movie.

### *MCNewAttachedController*

MCNewAttachedController takes an existing movie controller, associates a movie with it and attaches the controller to the movie. The controller is made visible if it was not already. If the controller is detached when the call is issued, it is first attached. The controller bounds rectangle is then offset such that its top left corner is aligned with the point specified in the call.

### *MCSetMovie*

MCSetMovie takes an existing controller and associates a new movie with it. The controller bounds rectangle is then offset such that its top left corner is aligned with the point specified in the call.

### *MCGetControllerBoundsRect*

The function for retrieving the bounds rectangle is MCGetControllerBoundsRect, which fills a Windows `RECT` structure with the desired coordinates:

```
RECT rcBounds;
MovieController mcController;
•
•
MCGetControllerBoundsRect (mcController, &rcBounds);
```

### *GetMovieBox*

You can always use GetMovieBox to obtain the coordinates of the movie only:

```
RECT rcMovie;
Movie mMovie;
•
•
GetMovieBox (mMovie, &rcMovie);
```

If no video information is available (see GetVideoInfo), the rectangle specified to receive the coordinates is made empty.

*Note: All QuickTime for Windows routines referencing a RECT or POINT assume client device coordinates.*

## 11. Movie Controller Attributes

Aside from features like attachment, activation state, size and position, movie controllers have other important attributes which can be manipulated by an application. Some of these attributes are stored in data structures which you can access as flags arranged in bit fields. Others are retrieved or set individually.

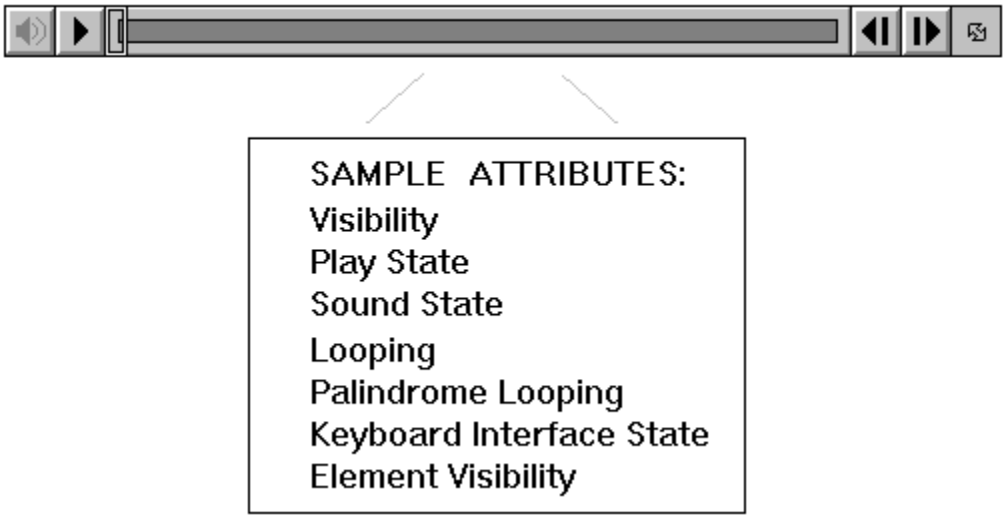



Figure 5. Movie Controller Attributes

If a movie controller needs to be hidden, for example, the easiest way to do it is to call the routine MCSetVisible (using `FALSE` makes the controller invisible):

```
BOOL bVisible;

MovieController mcController;

•

•

MCSetVisible (mcController, bVisible);
```

Invisible movie controllers may be attached, detached, active or inactive. You just can't see them. It is possible, however, to control a movie if its controller is not visible. For instance, you can stop or start a movie by single- or double-clicking (respectively) directly on it.

Also, you can use a movie controller's keyboard interface (if enabled) to stop, start or otherwise manipulate a movie. Finally, you can control a movie programatically using appropriate routines from the QuickTime for Windows API.

To query the visibility state of a movie controller, you can use the corresponding routine MCGetVisible. Setting visibility might be useful in applications handling multiple movies, special case movies and overall application aesthetics, just as you would detachment or activation.

The states of the Movie Controller's individual control elements are also considered attributes. To hide the speaker button or the left and right step buttons, you can use MCDoAction:

```
MovieController mcController;

•

•
```

```
MCDoAction (mcController, mcActionSetFlags,
   mcFlagSuppressStepButtons);
MCDoAction (mcController, mcActionSetFlags,
   mcFlagSuppressSpeakerButton);
```

To hide the grow box, you have to fill a Windows `RECT` structure with zeros, then pass its address to MCDoAction to use in setting the grow box bounds:

```
MovieController mcController;
RECT rcGrowBoxRect;
•
•
SetRectEmpty (&rcGrowBoxRect);
MCDoAction (mcController, mcActionSetGrowBoxBounds, &rcGrowBoxRect);
```

Enabling the keyboard interface for a movie controller is also done with MCDoAction, as is querying the state of a controller's keyboard interface:

```
MovieController mcController;
BOOL bActive;
•
•
MCDoAction (mcController, mcActionSetKeysEnabled, TRUE);
•
•
MCDoAction (mcController, mcActionGetKeysEnabled, &bActive);
```

If a movie controller's keyboard interface is enabled, the controller will accept keyboard input even if it has been set to the inactive state. Multiple controllers will receive the same keystrokes if their keyboard interfaces are enabled.

If you need to get more low-level information about a movie controller, the function MCGetControllerInfo is available. This call retrieves a long integer with bit flags denoting controller attributes such as whether the movie is playing, looping, looping back and forth, if the movie has sound, and so forth.

```
MovieController mcController;
LONG lMCInfoFlags;
•
•
MCGetControllerInfo (mcController, &lMCInfoFlags);
if (lMCInfoFlags & mcInfoHasSound)
   {
   •
   •
   }
```

# 12. Badges

A *badge* is a visual element displayed on the face of a movie to distinguish it from a static graphic when its movie controller is not visible. To be able to display a badge automatically, a movie controller must be created with the `mcWithBadge` creation flag.

Three conditions have to be met before a badge can be displayed automatically. First, the movie cannot be playing. Second, the badge flag must have been turned on when the movie controller was created (or with mcActionSetUseBadge). Third, your application must call MCSetVisible with `FALSE` as the second parameter, to make the movie controller invisible.

If the first two conditions are satisfied, calling MCSetVisible with `FALSE` (or creating the controller with `mcNotVisible`) hides the controller and causes the badge to be displayed.

```
Movie mMovie;

MovieController mcController;

RECT rcMovie;

•

•

mcController = NewMovieController (mMovie, &rcMovie, mcWithBadge, hWnd);
```

If a movie controller is displaying a badge, clicking the badge hides it and restores the movie controller (if the `mcWithBadge` flag is on).

*A good point to remember is that badge visibility is not an attribute of a movie controller, while the ability to* display *a badge is. Although you can specify that ability when the controller is created, you cannot use MCGetControllerInfo to query it.*

If your application needs more control over displaying badges, you can use the function MCDrawBadge. This routine lets you display a badge at any time, regardless of whether `mcWithBadge` is on or the movie is playing. Calling the function does not affect the state of the `mcWithBadge` flag.

When you call MCDrawBadge, you must set the second parameter to NULL. The third parameter receives the address of a handle to a badge region, which your program can use later at its discretion.

```
MovieController mcController;

HRGN hrgnBadge;

•

•

MCDrawBadge (mcController, NULL, &hrgnBadge);
```

Obviously, under certain circumstances you can create a situation where both a badge and a movie controller are visible at once, which is not good QuickTime for Windows style.

## 13. Actions and Filters

The function MCDoAction is one of the most versatile in the QuickTime for Windows API. Although it is available to you for handling specific, low-level tasks, it is also used by various high-level functions in QuickTime for Windows. Along with a movie controller object, it takes parameters for the action desired and additional data specific to that action, often the address of a boolean value denoting whether the action item should be toggled on or off:

```
MovieController mcController;
BOOL bFlag;
•
•
MCDoAction (mcController, mcActionActivate, &bFlag);
```

As we have seen, MCDoAction can be used to do things like starting a movie and setting a movie's activation state. Many other actions can be effected by this routine, however, and it is worth exploring them all to get a sense of the power and flexibility that MCDoAction provides.

Closely related to MCDoAction is the function MCSetActionFilter, which gives you a way to intercept the MCDoAction call. The usefulness of this routine is hard to underestimate, since QuickTime for Windows itself uses MCDoAction so extensively--especially in processing user interaction.

For example, almost anywhere you click on the movie controller generates a MCDoAction call internally. By creating carefully-designed filter functions, you can customize the behavior of your movie controllers to almost any level you wish.

MCSetActionFilter inserts the address of a user-defined filter function in the movie controller's data structure. This filter function is called automatically when your program calls MCDoAction. MCSetActionFilter's last parameter is a `LONG` which can be used to pass additional information to the filter function (e.g. the address of a structure containing data necessary for complex processing).

```
BOOL CALLBACK __export MyFilter (MovieController, UINT, LONG);
MovieController mcController;
struct {...} *pData;
•
•
MCSetActionFilter (mcController, MyFilter, (LONG) pData);
```

If you compile your program using Borland *smart callbacks* or Microsoft's `-GEs` compiler option, or your filter function is in a dynamic link library, you do not need to use `MakeProcInstance` on your filter address before calling MCSetActionFilter.

If a filter function is used, it gets a chance to process the action item before the movie controller. Its return value must be a boolean: `TRUE` indicates that the controller doesn't have to handle it. `FALSE` tells the controller to complete any appropriate processing of the action item.

To remove a filter, you must call MCSetActionFilter with the filter function address set to NULL. Since a filter is essentially a *callback* function, it must be declared as `CALLBACK` and listed in the `EXPORTS` section of your .DEF file.

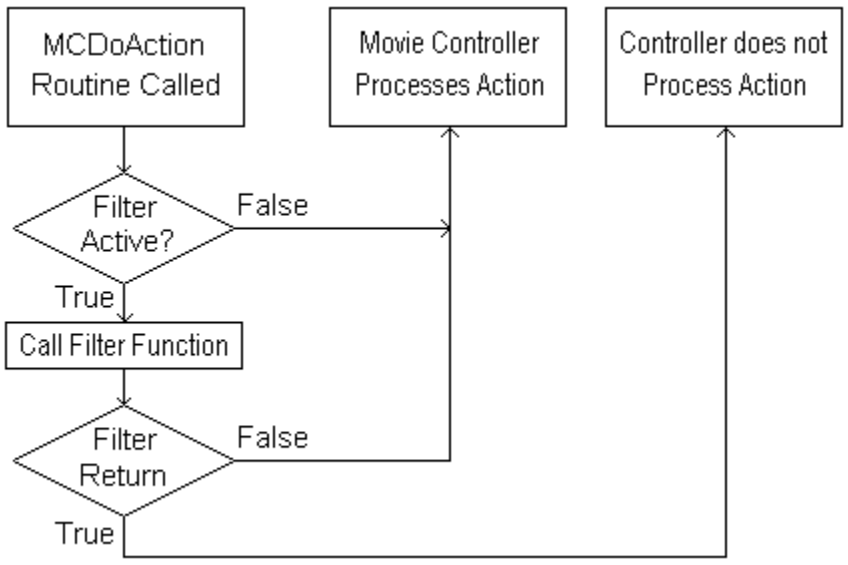



Figure 6. Using an Action Filter Function

You can view using an action filter as a kind of built-in subclassing. The following code fragment shows how you might set up your switch and case statements to handle a limited number of actions:

```
BOOL CALLBACK __export MyFilter (MovieController mcController,
   UINT uAction, LPVOID lpParam)
   {
   switch (uAction)
      {
      case mcActionDraw:
         •
         •
         return TRUE;
      case mcActionPlay:
         •
         •
         return TRUE;
      case mcActionKey:
         •
         •
         return TRUE;
      case mcActionBadgeClick:
         •
         •
         return TRUE;
      default:
         return FALSE;
```

```
        }
    }
```

# 14. Pictures

Like a movie, a QuickTime for Windows *picture* is a collection of data that can be rendered visually. Unlike a movie, a picture consists of a single complete image with no time coordinate system. This complete image is actually composed of one or more pieces, often arranged as bands within the area of the complete image.

Pictures are stored in picture files, from which they may be extracted using various QuickTime for Windows API routines and then displayed by your application. All of the pieces that comprise a complete image as described above are generally stored in the same picture file. Once extracted, a QuickTime for Windows picture is handled conceptually as a *picture object*, in a manner similar to a movie object.

QuickTime for Windows pictures are stored in the Macintosh PICT format (for a complete discussion of this format, refer to *Inside Mac Volumes V and VI*) or Macintosh JFIF format (see the document *JPEG File Interchange Format, Version 1.1*, available from C-Cube Microsystems, San Jose, CA 408-944-6335). Picture files and picture objects are manipulated by QuickTime for Windows API calls. For example, to extract a picture object:

```
PicHandle phPicture;

PicFile pfPicture;

•

•

if (OpenPictureFile ("PICTURE.PIC", &pfPicture, OF_READ) != noErr)
   {
   phPicture = GetPictureFromFile (pfPicture);

   ClosePictureFile (pfPicture);
   }

•

•

DisposePicture (phPicture);
```

As noted earlier, your QuickTime for Windows applications do not have to call EnterMovies if they are only going to deal with picture objects. QTInitialize is required, however, along with QTTerminate. Since picture objects occupy memory, they must be disposed of properly with DisposePicture (or its equivalent, KillPicture) when they are no longer needed. As with movies, a picture file should be closed as soon as possible once its picture is extracted.

The Macintosh PICT file format defines numerous *opcodes*, in much the same way as, for example, the TIFF format. Under QuickTime for Windows, however, only a subset of these opcodes are processed:

- `0x0090` - BitsRect
- `0x0091` - BitsRgn
- `0x0098` - PackBitsRect
- `0x0099` - PackBitsRgn
- `0x009A` - DirectBitsRect (denotes a direct image)
- `0x009B` - DirectBitsRgn (denotes a direct image)
- `0x8200` - Compressed QuickTime image
- `0x8201` - Uncompressed QuickTime image
- `0x0011` - Version

To draw the image contained in a picture object, you can use DrawPicture:

```
PicHandle phPicture;
HDC hdc;
RECT rcPicture;
•
•
DrawPicture (hdc, phPicture, &rcPicture, NULL);
```

Certain pictures may be stored with additional data defining a custom palette. You can extract this palette with GetPicturePalette and then use it in your Windows application to obtain a more faithful rendering of a picture:

```
PicHandle phPicture;
HDC hdc;
HPALETTE hpalPicture
RECT rcPicture;
•
•
// Standard Windows call to see if driver can handle a palette
   if (GetDeviceCaps (hdc, RASTERCAPS) || RC_PALETTE)
      {
      hpalPicture = GetPicturePalette (phPicture);
      SelectPalette (hdc, hpalPicture,0);
      RealizePalette (hdc);
      }
   •
   •
   DrawPicture (hdc, phPicture, &rcPicture, NULL);
```

Picture files cannot be created or edited, but the images in them may be converted to formats for editing and saving under Windows. For example, the following code puts a device independent bitmap, derived from a QuickTime for Windows picture, on the Windows clipboard:

```
PicFile pfPicture
PicHandle phPicture;
DIBHandle hdPicture;
•
•
// Extract a picture and convert it to Windows Device Independent
// Bitmap (DIB)
   if (OpenPictureFile ("PICTURE.PIC", &pfPicture, OF_READ) != noErr)
      {
      phPicture = GetPictureFromFile (pfPicture);
      ClosePictureFile (pfPicture);
      }
```

```
    •
    •
    hdPicture = PictureToDIB (phPicture);
    DisposePicture (phPicture);
// Put the DIB in the clipboard
    OpenClipboard (hWnd);
    EmptyClipboard ();
    SetClipboardData (cf_DIB, hdPicture);
    CloseClipboard ();
```

Some QuickTime for Windows API calls allow you to operate directly on a picture file without first extracting a picture object. For instance, DrawPictureFile draws the image contained in a file:

```
PicFile pfPicture;
RECT rcPict;
HDC hdc;
•
•
OpenPictureFile ("HOUSE.PIC", &pfPicture, OF_READ);
DrawPictureFile (hdc, pfPicture, &rcPict, NULL);
ClosePictureFile (pfPicture);
```

You can use GetPictureInfo to extract information about a picture object. Similarly, you can use GetPictureFileInfo to extract data directly from a picture file.

## 15. Getting Pictures from Movies

Movie data can be viewed as a collection of compressed still images. A routine that allows you to retrieve such individual images from a movie is GetMoviePict, which takes a specified movie time as a parameter.

MCGetCurrentTime retrieves the movie's current time, i.e. position on the movie's time axis. This function can be used whether a movie is playing or not.

```
Movie mMovie;
MovieController mcController;
PicHandle phMyPicHandle;
TimeValue tvTime;
TimeScale tsTime;
•
•
tvTime = MCGetCurrentTime (mcController, &tsTime);
phMyPicHandle = GetMoviePict (mMovie, tvTime);
```

The picture object obtained from GetMoviePict points to an image in a format unusable by Windows directly. If you want to convert it to a Windows format suitable for use by other Windows applications, your can do so using PictureToDIB. This routine returns a handle to a device-independent bitmap, which can then be used to put the picture in the Windows clipboard or send it to your printer.

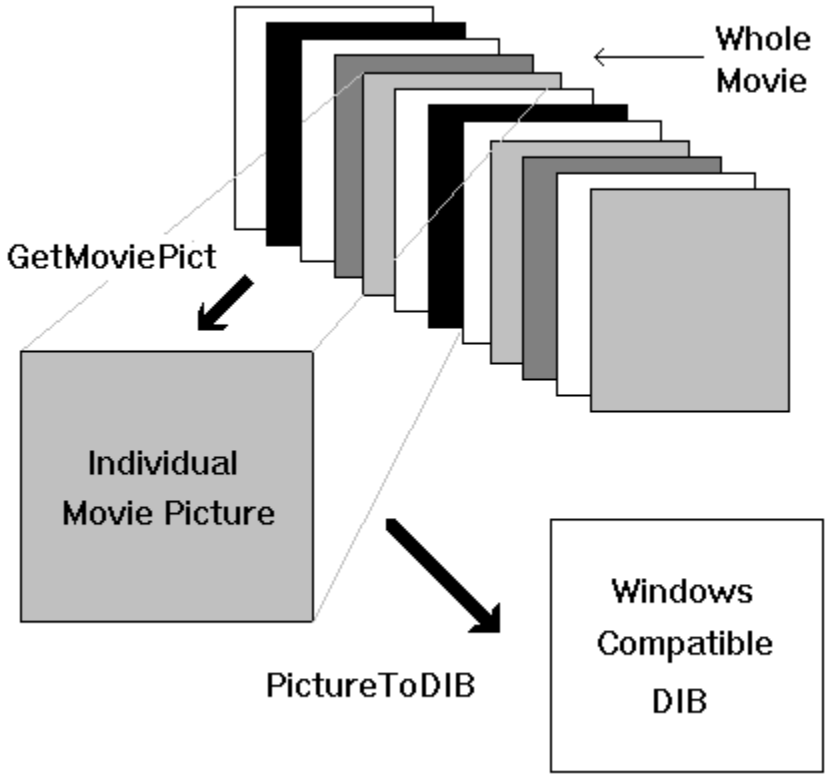



Figure 7. Retrieving a Picture from a Movie

The alternative to converting an image retrieved by GetMoviePict is to display it directly. Calling the function DrawPicture puts the picture on the screen at coordinates you specify. You'll need to supply a device context, the picture object reference and a display rectangle. Whatever you decide to do with a movie picture object you retrieve, you must free it when you are done with it.

```
Movie mMovie;
MovieController mcController;
PicHandle phMyPicHandle;
TimeValue tvTime;
•
•
tvTime = MCGetCurrentTime (mcController, /* Time scale address */);
phMyPicHandle = GetMoviePict (mMovie, tvTime);
DrawPicture (hdcMyDevCon, phMyPicHandle, &rcPicture, NULL);
•
•
DisposePicture (phMyPicHandle);
```

As with picture objects extracted from picture files, pictures extracted from movies may also contain custom palette information. You can use GetPicturePalette to retrieve this data and set the Windows palette to better render these individual movie images.

A *movie poster* is a frame in a movie selected when the movie was created to represent the movie when it is not loaded or not being played. You have access to this picture with GetMoviePosterPict, which returns an image object created from the frame designated as the movie's poster. One interesting way to use movie posters might be in an open movie dialog box. When the name of the movie is highlighted in the list box, its poster would be displayed next to it.

```
case LN_SELECT:
    •
    •
    OpenMovieFile (/* file name highlighted */, ...);
    NewMovieFromFile (...);
    phMyPicHandle = GetMoviePosterPict (/*NewMovieFromFile object */);
    hDIB = PictureToDIB (phMyPicHandle);
    /* Display DIB in dialog box using bitmap object. */
    break;
```

## 16. Getting User Data from Movies

User data is typically inserted into a movie by its creator to identify special characteristics, production credits, and so forth. Any movie can contain a *user data list*, which is available for use by your application. A user data list comprises all the user data for a movie, and may contain one or more *user data items*. Each user data item has several attributes:

- The type identifier - denotes the specific type of the item, e.g. date, copyright, etc.
- The index value - a unique, one-based number denoting list position among like types
- The data itself - generally text, possibly other data

To get a handle to a movie's user data, you call GetMovieUserData:

```
Movie mMovie;
UserData udData;
•
•
udData = GetMovieUserData (mMovie);
```

With this handle, you can parse the data. Each of the other functions which handle user data has a specific purpose in this regard:

GetNextUserDataType takes the user data handle and desired user data type as parameters. If the type parameter is 0, the routine returns the first user type in the user data list. For subsequent calls (for example, in a loop to get all the user data), use the previous value returned by this function. The current format of the user data type identifier in a QuickTime movie is four-character constant, which is supported in the Macintosh environment, but not directly under Windows. You can create the equivalent, however, with the macro QTFOURCC.

```
UserData udData;
OSType osType;
•
•
osType = QTFOURCC('©','d','a','y');
osType = GetNextUserDataType (udData, osType);
```

Below are some common user data types (note they are case sensitive). By convention, text user data types start with a "©" symbol. Remember to use the QTFOURCC macro.

`©cpy` Copyright statement

`©day` Date the movie's content was created

`©dir` Name of movie's director

`©ed1` to `©ed9` Edit dates and descriptions

`©fmt` Indication of movie format (computer-generated, digitized, etc.)

`©inf` Information about the movie

`©prd` Name of movie's producer

`©prd` Names of performers

`©req` Special hardware and software requirements

©src Credits for providers of movie source content

©wrt Name of movie's writer

LOOP Denotes that the movie expects to be played in loop mode. If the value of this user data type is empty or 0, normal loop mode is indicated. A value of 1 denotes palindrome loop mode.

WLOC Denotes that the last known position of the movie on the desktop is available, represented by two 16-bit integers contained in its associated value. Because movies are created on the Mac, this may not translate well to the Windows desktop.

CountUserDataType returns the number of items of a given type in a user data list. You pass it the handle to the user data list and the desired type:

```
UserData udData;
LONG lItemCount;
•
•
lItemCount = CountUserDataType (udData, QTFOURCC('©','d','a','y'));
```

GetUserData retrieves a specified user data item. You need to pass it the handle of a global memory block you have allocated, in which it will place the requested item. When you allocate the memory block, you should make it of an arbitrary size, since QuickTime for Windows will reallocate memory internally based on your handle if the data item requested is too big. You must free this handle explicitly when you are done with it.

In addition to the memory handle, you must also pass GetUserData the index value of the data item you want, and the address of a LONG which it fills with the size of the data item requested (in bytes).

```
UserData udData;
HGLOBAL ghMem;
LONG lIndex, lByteCount;
struct {...} *pData;
•
•
// Note arbitrary size of allocation request
if ((ghMem = GlobalAlloc (GMEM_MOVEABLE, 128)) == NULL);
   {
   /* Inform user of failure. */
   return;
   }
GetUserData (udData, &ghMem, QTFOURCC('t','e','s','t'), lIndex,
   &lByteCount);
pData = GlobalLock (ghMem);
•
•
/* Do something with user data item. */
•
•
```

```
GlobalUnlock (ghMem);
GlobalFree (ghMem);
```

When you specify a type of user data in this routine, you must know its format in advance. One way to handle this is to have `GlobalLock` return a pointer to a structure type you declare which maps onto the structure of the user data type you are retrieving.

GetUserDataText retrieves the text associated with a particular user data text item. Its parameters are the same as for GetUserData, with one exception: the region code. A region code is a value representing a particular language or country.

```
UserData udData;
LONG lIndex, lByteCount;
HGLOBAL ghMem;
LPSTR lpstrText;
•
•
ghMem = GlobalAlloc (GMEM_MOVEABLE, 128); // Note arbitrary size
GetUserDataText  (udData, &ghMem, QTFOURCC('@','d','a','y'),
   lIndex, 0, &lByteCount);
lpstrText = (LPSTR) GlobalLock (ghMem);
lpstrText [lByteCount] = '\0';
•
•
/* Do something with text string. */
•
•
GlobalUnlock (ghMem);
GlobalFree (ghMem);
```

In this example, 0 is the code for US (English).

# 17. Getting System Data from Movies

In addition to individual picture frames and user data, movies contain a substantial amount of other data that your QuickTime for Windows programs can make use of, such as preferred play settings, time-based information and so forth.

Preferred settings are data elements held by a movie that denote optimum performance characteristics. When a movie is created, the author has the opportunity to encode what he or she feels is the most suitable volume, play rate, etc., which can later be used to play the movie as the author intended.

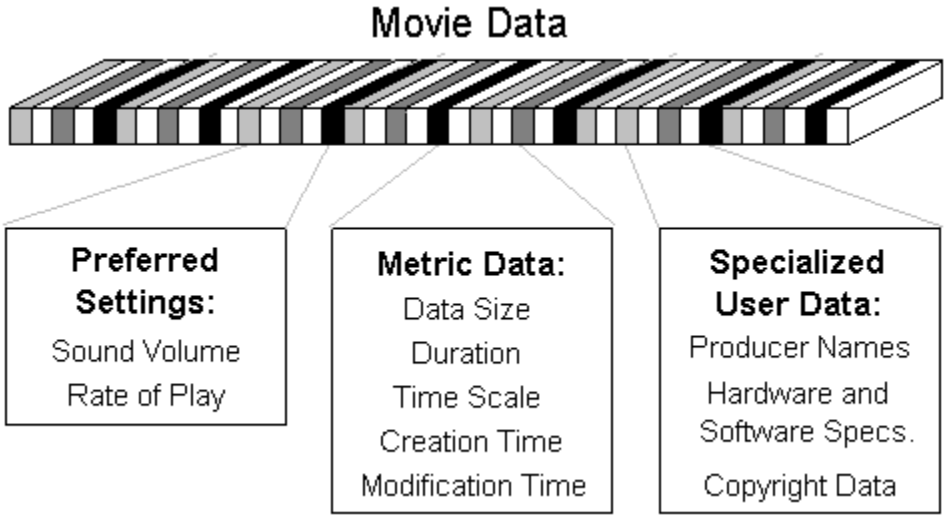



Figure 8. Available Movie System and User Data

For example, you can get the preferred volume with GetMoviePreferredVolume, then use the return value to set the movie volume with a call to MCDoAction with the mcActionSetVolume parameter.

To retrieve the preferred play rate, the call is GetMoviePreferredRate. You can set the movie's play rate as above using the mcActionPlay action with the returned rate as the additional parameter.

The second category, metric data, is more diverse. You will be the best judge of how to use these particular routines in your QuickTime for Windows programs. The routine GetMovieDataSize, for instance, returns (in bytes) the size of a specified movie segment, including sound.

GetMovieTimeScale returns the movie's time scale, which (as we noted earlier) is a specific fraction of a second. GetMovieDuration returns a movie's duration expressed in terms of its time scale.

You can manipulate a movie's time scale with ConvertTimeScale. The timestamp functions, GetMovieCreationTime and GetMovieModificationTime, return the values for when the movie was created and last modified, respectively.

# 18. Cover Procedures

QuickTime for Windows allows your application to perform custom processing whenever one of your movies covers a screen region or reveals a region that was previously covered. You perform this processing in cover procedures. Cover procedures are useful in handling movies with "empty segments," i.e. portions of movies intentionally lacking any visual element.

By default, QuickTime for Windows will display the normal background color during an empty segment. You can use a cover procedure to display other information meaningful to your application.

There are two types of cover procedures: those that are called when your movie covers a screen region, and those called when it uncovers a screen region, revealing a region that was previously covered. Cover procedures that are called when your movie covers a screen region are responsible for erasing the region--you may choose to save the hidden region in a bitmap. Cover procedures that are called when your movie reveals a hidden region must redisplay the hidden region.

Use SetMovieCoverProcs to set both types of cover procedures. The following example shows how to establish a cover procedure called when your movie uncovers a screen region.

```
OSErr CALLBACK __export CoverProc (Movie, HDC, LONG);
•
•
HWND hWnd;
Movie mMovie;
•
•
SetMovieCoverProcs (mMovie, CoverProc, 5879);
•
•
OSErr CALLBACK __export CoverProc (Movie m, HDC hdc, lID)
   {
   RECT rcClip;
   GetClipBox (hdc, &rcClip;
   FillRect (hdc, &rcClip, GetStockObject (WHITE_BRUSH));
   return 0;
   }
```

Note that the third parameter to SetMovieCoverProcs is an arbitrary constant passed directly to your routine. You can use this to distinguish invocations when your cover procedure is shared by two or more movies.

If you compile your program using Borland *smart callbacks* or Microsoft's `-GEs` compiler option, or your filter function is in a dynamic link library, you do not need to use `MakeProcInstance` on your cover procedure address before calling `SetMovieCoverProcs`. Since a cover procedure is essentially a *callback* function, it must be declared as `CALLBACK` and listed in the `EXPORTS` section of your .DEF file.

# 19. QuickTime for Windows Error Handling

The QuickTime for Windows API provides two routines for trapping non-Movie Controller function errors: GetMoviesError and GetMoviesStickyError. Movie Controller functions do not return error conditions.

# B. The QuickTime for Windows Environment

# 1. Hardware Considerations

The supported environment for QuickTime for Windows is Windows 3.1, either standard or enhanced mode, running on an i386 or i486 machine. If a program incorporating QuickTime for Windows is run in a non-supported environment, QTInitialize will fail. If this happens, it is your responsibility not to execute any further QuickTime for Windows calls. QuickTime for Windows does provide some assistance in this area by making all of its calls no-ops when QTInitialize fails, but you should take the extra steps to not even call the functions in a non-QuickTime for Windows environment. Doing so will ensure that your applications continue to run normally when QuickTime for Windows is integrated, even when QuickTime for Windows cannot.

## 2. Developing QuickTime for Windows Programs

To start building QuickTime for Windows programs, you need to make four changes to your development environment and program source files:

- Include the library file QTW.LIB in the link line of your program's make file
- Add the line `#include QTW.H` to your program's source file
- Change the stack size to at least 16K in your program's .DEF file
- Check that the QuickTime for Windows installation program has updated the SET LIB, SET INCLUDE and PATH environment variables in your AUTOEXEC.BAT file to access all of the QuickTime for Windows development tools.

# 3. QuickTime for Windows On-line Help

If you have installed QuickTime for Windows from diskettes, all of the help files are in the directory `\qtw\help`. They are in the standard .HLP format, accessible with *Quick Help*. If you have installed from CD-ROM, you will have the standard .HLP files plus their source code files (with the extension .RTF) and their corresponding help project files (with the extension .HPJ), also in `\qtw\help`. Of particular note are the files for the Movie Controller, which you can integrate with your application's help system.

You can rebuild the compiled help files using the Windows help compiler. For example, to build the Movie Controller help file, you would invoke:

```
HC31 MCENU.HPJ
```

The three-letter "ENU" string in the file name indicates the U.S. English version. To compile help files for other languages, use the appropriate source files in `\qtw\help`.

# C. QuickTime for Windows Applications

QuickTime for Windows provides two sample applications for viewing QuickTime movies and pictures: *Movie Player* and *Picture Viewer*. These programs use the Microsoft standard Multiple Document Interface (MDI) to view multiple movies or pictures, respectively. Complete source code is provided for each application for use as a learning tool. When running either program, you will find extensive on-line help available through the Help menu item or the F1 function key.

## Related Topics:

# 1. The Movie Player

This application lets you play one or more movies in its main window. All movies run in standard MDI child windows. You can resize any of the movies by dragging on their borders, or by using the grow box in the lower right corner. Individual movie frames can be copied to the clipboard through the *Edit* menu item, and information about the movie is available under the *Movie* menu item. The Movie Player executable is in the `\qtw\bin` subdirectory. Its source code is in `\qtw\mplayer`. You can build PLAYER.EXE with the make file PLAYER.MAK (in standard NMAKE format), also located in this directory.

Online help files for the Movie Player are provided in two formats: PLAYENU.RTF (*rich text* format, only if you installed from CD-ROM) and PLAYENU.HLP (standard compiled help files, usable by the Windows help subsystem). These help files are in the directory `\qtw\help` and are currently localized for the U.S. English language. You can localize them for other languages at your discretion (no other localization is normally required for QuickTime for Windows programs). Help files for the Movie Controller, MCENU.RTF and MCENU.HLP, are in the same format and location.

## 2. The Picture Viewer

This application lets you view one or more pictures in its main window. All pictures are displayed in standard MDI child windows, which you can resize by dragging on their frame-sizing borders or by using the grow box in the lower right corner. Individual pictures can be copied to the clipboard through the *Edit* menu item, and information about the picture is available under the *Image* menu item. The Picture Viewer executable is in the `\qtw\bin` subdirectory. Its source code is in `\qtw\pviewer`. You can build VIEWER.EXE by executing the make file VIEWER.MAK (in standard NMAKE format), also located in this directory.

Online help files for the Picture Viewer are provided in two formats: VIEWENU.RTF (*rich text* format, only if you installed from CD-ROM) and VIEWENU.HLP (standard compiled help files, usable by the Windows help subsystem). These help files are in the directory `\qtw\help` and are currently localized for the U.S. English language. You can localize them for other languages at your discretion (no other localization is normally required for QuickTime for Windows programs).

# D. QuickTime for Windows vs. QuickTime for the Macintosh

# 1. Summary

As an experienced QuickTime programmer ready to use the QuickTime for Windows API, you know about differences between the Windows and Macintosh platforms. You should also be aware of how QuickTime and QuickTime for Windows themselves differ in implementation.

We noted earlier that QuickTime movies can be created and edited on the Macintosh, while they can be handled in playback mode only in the current version of QuickTime for Windows. It is also worth re-emphasizing that the primary focus of the QuickTime for Windows API and related documentation is the Movie Controller.

Although QuickTime for Windows' API is based as closely as possible on QuickTime's, the platform differences noted above have necessitated the creation of QuickTime for Windows calls with no counterpart on the Macintosh side. These are discussed in context in the material that follows. Equally important is that many of the QuickTime Toolbox routines available to the Macintosh developer are not exposed in the QuickTime for Windows API, since the focus is on the Movie Controller.

Finally, you should be aware of some additional QuickTime concepts not implemented or supported in this version of QuickTime for Windows:

- Tracks
- Media
- Components
- User-writable CODECs

## 2. The Movie Controller

The important ideas to keep in mind regarding the QuickTime for Windows Movie Controller are:

- Playing movies under QuickTime for Windows is possible only with the Movie Controller, as opposed to under QuickTime, which allows movies to be played using its Toolbox API.
- You cannot create a custom movie controller component.
- The QuickTime for Windows Movie Controller is functionally identical to the default movie controller under QuickTime.
- You can simulate the appearance of a QuickTime toolbox application using an invisible movie controller.

## 3. Initialization and Termination Differences

QuickTime is an operating system extension on the Macintosh and does not need to be explicitly initialized. Under QuickTime for Windows, any application that makes calls to the QuickTime for Windows libraries must first verify that the libraries are available on the system. This is accomplished with the new QuickTime for Windows-only routine QTInitialize, which establishes links to those libraries if they are indeed present. The bookend function, QTTerminate, also new to QuickTime for Windows, must be called before your QuickTime for Windows-enabled program is unloaded.

## 4. Picture Handling Differences

Since pictures on the Macintosh are also generally handled at the operating system level, there are a number of new routines to deal with individual QuickTime for Windows images.

ClosePictureFile

DisposePicture

DrawPicture

GetPictureFileInfo

GetPictureFromFile

GetPictureInfo

GetPicturePalette

KillPicture

OpenPictureFile

PictureToDIB

## 5. Other Differences

The following new routines are included in the QuickTime for Windows API to bridge other platform differences.

GetSoundInfo

GetVideoInfo

MAKELFIXED (macro)

MAKESFIXED (macro)

MCIsPlayerMessage (formerly `MCIsPlayerEvent)`

NormalizeRect

QTFOURCC (macro)

## 6. QuickTime API Calls Supported by QuickTime for Windows

**Application Defined Movie Routines**

SetMovieCoverProcs

**Disabling Movies and Tracks**

GetMovieActive

SetMovieActive

**Enhancing Movie Playback Performance**

PrerollMovie

**Error Routines**

ClearMoviesStickyError

GetMoviesError

GetMoviesStickyError

**Movies and the Event Loop**

GetMovieStatus

PtInMovie

UpdateMovie

**Generating Pictures from Movies**

GetMoviePict

GetMoviePosterPict

**Initializing the Movie Toolbox**

EnterMovies

ExitMovies

**Movie Controller**

DisposeMovieController

MCActivate

MCDoAction

MCDraw

MCDrawBadge

MCGetControllerBoundsRect

MCGetControllerInfo

MCGetCurrentTime

MCGetMovie

MCGetVisible

MCIdle

MCIsControllerAttached

MCIsPlayerMessage

MCKey

MCNewAttachedController

MCPositionController

MCSetActionFilter

MCSetControllerAttached

MCSetControllerBoundsRect

MCSetVisible

NewMovieController

**Determining Movie Creation and Modification Time**

GetMovieCreationTime

GetMovieDataSize

GetMovieModificationTime

**Movie Routines**

CloseMovieFile

DeleteMovieFile

DisposeMovie

GetMovieBox

NewMovieFromDataFork

NewMovieFromFile

OpenMovieFile

**Working with Pictures and Picture Files**

DisposePicture

DrawPictureFile

GetPictureFileHeader

KillPicture

**Movie Posters and Movie Previews**

GetMoviePosterTime

Preferred Movie Settings

GetMoviePreferredRate

GetMoviePreferredVolume

**Time Base Routines**

AddTime

ConvertTimeScale

SubtractTime

**Working with Movie User Data**

CountUserDataType

GetMovieUserData

GetNextUserDataType

GetUserData

GetUserDataText

**Working with Movie Time**

GetMovieActiveSegment

GetMovieDuration

GetMovieTime

GetMovieTimeScale

# E. Preparing Macintosh movie and picture files for QuickTime for Windows

QuickTime movies prepared on the Macintosh to play under Windows need to have two related characteristics. They must be 1) self-contained, and 2) contained in a single fork file. These characteristics are set by the Macintosh application that saves the movie. Such an application is the Movie Converter, which is part of the QuickTime Starter Kit (M1269LL/A) available from Apple Dealers everywhere.

Macintosh QuickTime pictures may be transferred to a Windows machine directly (e.g., over a network or with a Mac to PC file transfer program) and viewed without any special preparation.

To use Movie Converter to create a movie file that can be ported to a Windows machine:

1. Launch Movie Converter
2. Open the QuickTime movie to be saved.
3. In the File Menu select "Save As_".
4. Click the "Make movie self-contained" button. This creates a movie that contains no references to other files.
5. Check "Playable on non-Apple computers". This creates a movie file that does not depend on resources.
6. Save the file.

The file just created can now be ported to a Windows machine (e.g., over a network or with a Mac to PC file exchange program) and viewed with any application that supports QuickTime for Windows. More Macintosh applications are expected to support this saving mode in the near future.

# A. Introduction

The series of sample programs presented in this section of the on-line manual is intended as a learning tool. While they clearly demonstrate the power and flexibility of the QuickTime for Windows API, none of the programs should be taken out of context or used in production quality applications without careful consideration. Although the complete source code for each program is listed out in this section, the files are also in the `\qtw\samples` directory of your installed QuickTime for Windows environment.

# B. WINPLAY1 - Your First QuickTime for Windows Program

# 1. Introduction

WINPLAY1 serves one purpose: it puts into context the essential steps for initializing, executing and disposing various QuickTime for Windows API components required to play a movie. Its user interface is a plain frame window completely filled by a single movie and attached movie controller.

## 2. The WINPLAY1 Source Code

- WINPLAY1.MAK is the standard make file.
- WINPLAY1.DEF is the module definition file.
- WINPLAY1.C is the C source file.

```
WINPLAY1.MAK

ALL : WINPLAY1.EXE

WINPLAY1.OBJ : WINPLAY1.C
 cl -c -AS -DSTRICT -G2 -Zpel -W3 -WX -Od winplay1.c

WINPLAY1.EXE : WINPLAY1.OBJ WINPLAY1.DEF
 link /nod /a:16 winplay1, winplay1.exe, nul, qtw libw slibcew, \
    winplay1.def;
 rc winplay1.exe


WINPLAY1.DEF
NAME        WINPLAY1
DESCRIPTION 'Sample Application'
EXETYPE     WINDOWS
STUB        'winstub.exe'
CODE        PRELOAD MOVEABLE DISCARDABLE
DATA        PRELOAD MOVEABLE MULTIPLE
HEAPSIZE    1024
STACKSIZE   16384


WINPLAY1.C

#include <windows.h>
#include <qtw.h>

long FAR PASCAL __export WndProc (HWND, UINT, WPARAM, LPARAM);

MovieFile mfMovie;
RECT rcMovie;
Movie mMovie;
MovieController mcController;

int PASCAL WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
   LPSTR lpszCmdParam, int nCmdShow)
   {
   static char szAppName[] = "WinPlay1";
   HWND        hWnd;
   MSG         msg;
   WNDCLASS    wndclass;

// Establish links to QuickTime for Windows
   if (QTInitialize (NULL))
      {
      MessageBox (NULL, "QTInitialize failure", szAppName, MB_OK);
      return 0;
```

```
      }
// Allocate memory required for playing movies
   if (EnterMovies ())
      {
      MessageBox (NULL, "EnterMovies failure", szAppName, MB_OK);
      return 0;
      }
// Register and create main window
   if (!hPrevInstance)
      {
      wndclass.style         = CS_DBLCLKS | CS_HREDRAW | CS_VREDRAW;
      wndclass.lpfnWndProc   = WndProc;
      wndclass.cbClsExtra    = 0;
      wndclass.cbWndExtra    = 0;
      wndclass.hInstance     = hInstance;
      wndclass.hIcon         = LoadIcon (NULL,IDI_APPLICATION);
      wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW);
      wndclass.hbrBackground = (HBRUSH) (COLOR_WINDOW + 1);
      wndclass.lpszMenuName  = NULL;
      wndclass.lpszClassName = szAppName;
      if (!RegisterClass (&wndclass))
         {
         MessageBox (NULL, "RegisterClass failure", szAppName, MB_OK);
         return 0;
         }
      }
   hWnd = CreateWindow (szAppName, szAppName, WS_CAPTION | WS_SYSMENU |
      WS_CLIPCHILDREN | WS_OVERLAPPED, CW_USEDEFAULT, CW_USEDEFAULT,
      CW_USEDEFAULT, CW_USEDEFAULT, NULL, NULL, hInstance, NULL);
   if (hWnd == NULL)
      {
      MessageBox (NULL, "CreateWindow failure", szAppName, MB_OK);
      return 0;
      }
// Instantiate the movie
   if (OpenMovieFile ("SAMPLE.MOV", &mfMovie, OF_READ) != noErr)
      {
      MessageBox (NULL, "OpenMovieFile failure", szAppName, MB_OK);
      return 0;
      }
   NewMovieFromFile (&mMovie, mfMovie, NULL, NULL, 0, NULL);
   CloseMovieFile (mfMovie);
// Instantiate the movie controller
   GetMovieBox (mMovie, &rcMovie);
   OffsetRect(&rcMovie, -rcMovie.left, -rcMovie.top);
   mcController = NewMovieController (mMovie, &rcMovie,
      mcTopLeftMovie + mcScaleMovieToFit, hWnd);
// Make the movie paused initially
   MCDoAction (mcController, mcActionPlay, 0);
// Eliminate the grow box
   SetRectEmpty (&rcMovie);
   MCDoAction (mcController, mcActionSetGrowBoxBounds, &rcMovie);
// Make the frame just big enough for the movie
   MCGetControllerBoundsRect (mcController, &rcMovie);
   AdjustWindowRect (&rcMovie, WS_CAPTION | WS_OVERLAPPED, FALSE);
   OffsetRect(&rcMovie, -rcMovie.left, -rcMovie.top);
   SetWindowPos (hWnd, 0, 0, 0,
```

```
      rcMovie.right, rcMovie.bottom, SWP_NOMOVE | SWP_NOZORDER);
// Make the movie active
   SetMovieActive (mMovie, TRUE);
// Make the main movie visible
   ShowWindow (hWnd, nCmdShow);
   UpdateWindow (hWnd);
// Play the movie
   while (GetMessage (&msg, NULL, 0, 0))
      {
      TranslateMessage (&msg);
      DispatchMessage (&msg);
      }
// Destroy the movie controller
   DisposeMovieController (mcController);
// Destroy the movie
   DisposeMovie (mMovie);
// Cut the connections to QuickTime for Windows
   ExitMovies ();
   QTTerminate ();
// Return to Windows
   return msg.wParam;
   }

long FAR PASCAL __export WndProc (HWND hWnd, UINT message,
   WPARAM wParam, LPARAM lParam)
   {
   PAINTSTRUCT ps;
// Drive the movie controller
   if (MCIsPlayerMessage (mcController, hWnd, message, wParam, lParam))
      return 0;
// Process the windows message
   switch (message)
      {
      case WM_PAINT:
         if (!BeginPaint (hWnd, &ps))
            return 0;
         EndPaint (hWnd, &ps);
         return 0;
      case WM_DESTROY:
         PostQuitMessage (0);
         return 0;
      }
// Return to Windows
   return DefWindowProc (hWnd, message, wParam, lParam);
   }
```

## 3. Building QuickTime for Windows Programs

The most significant difference between WINPLAY1.MAK and an otherwise standard make file is in the link line: a file named QTW.LIB is specified in the library list. In general, the only change necessary for your existing Windows make files is to make sure QTW.LIB is added to your list of statically-linked libraries.

WINPLAY1.DEF is provided only to complete the source file set for this tutorial. Module definition files for your existing Windows programs generally will not have to be modified for QuickTime for Windows.

# 4. Initializing QuickTime for Windows Programs

The first QuickTime for Windows function in WINPLAY1.C is QTInitialize, which has a void parameter list and returns one of five possible values:

| | |
|---|---|
| `QTI_OK` | Success |
| `QT_FAIL_CORRUPTDLL` | A QuickTime for Windows DLL failed to load |
| `QTI_FAIL_NOEXIST` | QuickTime for Windows is not installed |
| `QTI_FAIL_286` | QuickTime for Windows requires a 386 or better |
| `QTI_FAIL_WIN30` | Windows 3.1 or better required |

This routine must be called before any other QuickTime for Windows function. Although it is performed automatically when any such function is executed, you should call it explicitly as a matter of programming style. Its primary purpose is to bind QuickTime for Windows-enabled applications to QuickTime for Windows at *run time.* Normally, a program utilizing DLLs is bound to them at link time; if calls to the DLLs are not resolved at load time, the program fails to load. The function QTInitialize provides *access* to QuickTime for Windows functions after the program has loaded. If QuickTime for Windows is not installed, the program will fail to play movies but otherwise run normally.

For instance, if you were the developer of an existing word processing program, you might want to add the ability to play movies in your documents but not cripple the product because it failed to load on a non-QuickTime for Windows system. QuickTime for Windows will accommodate you on both accounts. In effect, you can develop a QuickTime for Windows-enabled application without worrying about whether its DLLs will be present on future host systems.

QTInitialize also provides safety features to prevent a fatal failure if the application is running on a non-supported platform, or if the application accidentally makes a QuickTime for Windows call when QuickTime for Windows is not present. In these cases, all QuickTime for Windows calls are no-ops.

In WINPLAY1, a standard Windows message box is displayed if QTInitialize does not return QTI_OK, and the program exits when the message box is dismissed. If we fell through to the rest of the QuickTime for Windows functions, each of them would return unsuccessfully and no movie would be displayed. The program's main window would be created, however, and it would behave normally.

If QTInitialize returns successfully, the program calls EnterMovies to allocate memory required by QuickTime for Windows (not its movies), including the internal scheduler tables, etc., that will be used to track movies for this program. EnterMovies has a void parameter list and returns an `OSErr`. An `OSErr` is returned by a number of QuickTime for Windows functions. 0 indicates no error. Various other integer values denote QuickTime for Windows error conditions which your program may react to as you deem appropriate. Please see Appendix A for a breakout of these error codes.

WINPLAY1 checks the return and puts up a message box, followed by a program exit, if an error condition is indicated. An application may call EnterMovies multiple times, but memory will be allocated only for the first call.

As noted in the overview, QTInitialize and EnterMovies (if your program plays movies) only need to be called once during the life of your QuickTime for Windows application. Functions which deal with initializing individual movies, discussed next, need to be executed for each QuickTime for Windows movie your program incorporates.

## 5. Loading a Movie

Assuming WINPLAY1 has been successfully initialized for using the QuickTime for Windows libraries, it can now proceed to ready a specific movie for playing. OpenMovieFile is hardcoded to open the movie file SAMPLE.MOV, its first parameter. Its second parameter is the address of `mfMovie`, which will be passed to NewMovieFromFile.

The third parameter is an integer expressed as a standard file open flag as defined for the Windows `OpenFile` function, normally OF_READ, since movies generally cannot be opened other than read-only in the current version of QuickTime for Windows. OpenMovieFile returns an `OSErr`, which is checked and handled in the same way as it was for EnterMovies and QTInitialize.

*Note: For overall clarity, return codes are not checked for QuickTime for Windows functions beyond this point. Of course, in production-grade code all QuickTime for Windows return values would be checked and handled appropriately.*

---

To initialize a movie object to pass to NewMovieController, we have to call NewMovieFromFile. Its first parameter is the address of our movie object `mMovie`. Second is the `mfMovie` assigned by QuickTime for Windows when we called OpenMovieFile. The fifth parameter is hardcoded to `0` to mark it simply as inactive. The rest of the parameters are set to NULL in the current version of QuickTime for Windows. For each movie you want to play, you must call OpenMovieFile and NewMovieFromFile. WINPLAY1 only plays a single movie, and thus only makes the calls once.

CloseMovieFile is called next, since movie files should not be left open any longer than necessary. It takes the popular `mfMovie` as its only parameter.

## 6. Creating a Movie Controller

While NewMovieFromFile allocates and initializes all storage required for the movie and performs various internal tasks (e.g. telling QuickTime for Windows' scheduler to add the movie to its tables), there is still some conceptual distance to go before show time. What we have now is access to a sizable collection of movie data with no mechanism to play it. As explained in the overview, this is the role of the Movie Controller.

Before bringing up QuickTime for Windows' heavy artillery, we must first package up some data to pass it, specifically our movie's size and position within WINPLAY1's client area. The routine GetMovieBox provides these values, which are the natural dimensions of the movie as contained in the movie file (if the movie is freshly extracted with NewMovieFromFile).

We are now prepared to call NewMovieController, which must be done for each movie controller you wish to create (again, our sample program only has one, thus a single call). The parameters are:

- `mMovie`, the movie object assigned by QuickTime for Windows when it processed NewMovieFromFile
- the address of `rcMovie`, the  structure we have just filled with our movie's desired dimensions and coordinates
- `mcTopLeftMovie` and `mcScaleMovieToFit`, standard controller creation flags for displaying the movie in the movie rectangle (`rcMovie`)
- `hWnd`, the window handle for WINPLAY1, whose window will be the parent for the new movie controller and associated movie.

NewMovieController returns a `MovieController` object, an entity which you will use extensively in subsequent QuickTime for Windows calls.

Several key things now happen involving the QuickTime for Windows internal functions and data structures. The visible effect, once the movie is made visible, is the creation of the movie controller and its individual controls.

Before we call `ShowWindow`, however, we have to make WINPLAY1's frame window just big enough to enclose the movie and movie controller. This is accomplished with a combination of Windows calls and the routine MCGetControllerBoundsRect.

As explained in the overview, once a movie is associated with a controller, it starts playing immediately (assuming it has a non-zero play rate, which is normally the case). To make a movie paused when first visible and associated with a new controller, you can use MCDoAction with an action of mcActionPlay and a play rate of 0. It is good style to do this as soon as possible after performing the association.

It is important to note again that movies and movie controllers are not married for life. Movie controllers can be dynamically reassigned to movies at any point in your program, providing they are properly initialized. Destroying one does not destroy the other, nor does disconnecting a movie/movie controller pair disable either component. You will learn various ways to exploit this feature as you explore this tutorial.

## 7. Modifying the Window Procedure

The single piece of QuickTime for Windows code in `WndProc` is the routine MCIsPlayerMessage, but it wields significant power. Its parameters are:

- `mcController`, the movie controller object initialized in NewMovieController
- `hWnd`, the main window handle of WINPLAY1
- `message`, `wParam` and `lParam`, the same parameters passed in to `WndProc`.

To elaborate on the overview, the job of MCIsPlayerMessage is to redirect all messages targeted for the movie controller. If a message received by `WndProc` is not meant for the movie controller, MCIsPlayerMessage returns 0 and the message gets processed normally. If the message is supposed to be handled by the movie controller, MCIsPlayerMessage returns non-zero and the message does not get `switch`ed.

*Remember that for each movie controller you create, you have to code a separate call to* `MCIsPlayerMessage` *with the corresponding* `mcController` *variable as the first parameter. Since WINPLAY1 creates a lone controller, we only make the call once.*

## 8. Cleaning Up

Before WINPLAY1 exits, it needs to make sure it has not left any garbage lying around or kept any resources tied up. We do this in three stages, conceptually the reverse order of how the initialization was handled. First, we destroy the movie controller by calling DisposeMovieController, which takes the `mcController` object as its only parameter, and needs to be called for every movie controller you have created.

Second, the movie is released by executing DisposeMovie. This, too, is required for each movie you have instantiated, with the appropriate `mMovie` object as its sole parameter. Finally, ExitMovies (if your application plays movies) and QTTerminate are invoked. Like their counterparts that handle QuickTime for Windows initialization, they must only be called once by your program. As noted in the overview, executing QTInitialize is not required, but is recommended for good overall style.

*Remember that while destroying a window with a movie controller in it causes the function DisposeMovieController to be called internally for that controller, this is a safety feature only. Good QuickTime for Windows style dictates specifically disposing the controller.*

## 9. Running WINPLAY1.EXE

Having successfully compiled and linked WINPLAY1.EXE, you will want to fire it up and watch it play a movie. Before you do, however, you need to check that the movie name hardcoded in the OpenMovieFile routine matches the file name and location of the movie you expect to play. Since WINPLAY1.EXE only specifies the movie name (and not the path), make sure SAMPLE.MOV is in the same directory as WINPLAY1.EXE before you run it. If you want to play other movies without rebuilding WINPLAY1.EXE, you can copy any other sample movie files to the directory containing WINPLAY1.EXE, using the hardcoded movie name as a target file name.

Once you have made sure WINPLAY1.EXE can find its data, you should try to run it, preferably using the **Run** option under the Program Manager's **File** menu item. Clicking on the face of the movie window or the start button in the movie controller will run the movie. Now is probably a good time to experiment with the other movie controller buttons to get a feel for its basic operation.

# C. STEREO - Managing Multiple Movies

# 1. Introduction

Now that you can play a movie in a Windows program, you should next understand the issues of dealing with various movies in the same application. In this section, you will create a program called STEREO.EXE which plays two movies simultaneously and lets you dynamically detach their controllers. The concepts we'll explore include:

- Active and inactive states of movies and movie controllers
- Attached and detached movie controllers
- Resizing movies and movie controllers
- Multiple calls to MCIsPlayerMessage in a window procedure.

## 2. The STEREO Source Code

Before getting into the STEREO.C listing, you should note that the Common Dialog Box Library is used to create the Open Movie dialog box. COMMDLG.LIB is included on the link line of STEREO.MAK.

```
STEREO.MAK

ALL : STEREO.EXE

STEREO.OBJ : STEREO.C STEREO.H
 cl -c -AS -DSTRICT -G2 -Zpel -W3 -WX -Od stereo.c

STEREO.RES : STEREO.RC STEREO.H
 rc -r stereo.rc

STEREO.EXE : STEREO.OBJ STEREO.RES STEREO.DEF
 link /nod /a:16 stereo, stereo.exe, nul, qtw commdlg libw slibcew, \
    stereo.def;
 rc stereo.res


STEREO.DEF

NAME        STEREO
DESCRIPTION 'Sample Application'
EXETYPE     WINDOWS
STUB        'winstub.exe'
CODE        PRELOAD MOVEABLE DISCARDABLE
DATA        PRELOAD MOVEABLE MULTIPLE
HEAPSIZE    1024
STACKSIZE   16384


STEREO.H
#define IDM_OPEN          1
#define IDM_ATTACH        2
#define IDM_DETACH        3


STEREO.RC

#include <windows.h>
#include "stereo.h"

stereo MENU
  {
  POPUP "&File"
    {
    MENUITEM "&Open...", IDM_OPEN
    }
  POPUP "&Action"
    {
    MENUITEM "&Attach Controller", IDM_ATTACH
    MENUITEM "&Detach Controller", IDM_DETACH
    }
  }
```

STEREO.C

```
#include <windows.h>
#include <commdlg.h>
#include <string.h>
#include <stdlib.h>
#include <direct.h>
#include <qtw.h>
#include "stereo.h"

#ifdef __BORLANDC__
   #define _getcwd getcwd
#endif

long FAR PASCAL __export WndProc (HWND, UINT, WPARAM, LPARAM);
VOID CalcSize (HWND);

RECT rcLeft, rcRight, rcMovieBox, rcClient;
MovieController mcLeft, mcRight, mcActive;

int PASCAL WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
   LPSTR lpszCmdParam, int nCmdShow)
   {
   static char szAppName [] = "Stereo";
   HWND        hWnd;
   MSG         msg;
   WNDCLASS    wndclass;
// Establish links to QuickTime for Windows
   if (QTInitialize (NULL))
      {
      MessageBox (NULL, "QTInitialize failure", szAppName, MB_OK);
      return 0;
      }
// Allocate memory required for playing movies
   if (EnterMovies ())
      {
      MessageBox (NULL, "EnterMovies failure", szAppName, MB_OK);
      return 0;
      }
// Register and create main window
   if (!hPrevInstance)
      {
      wndclass.style         = CS_DBLCLKS | CS_HREDRAW | CS_VREDRAW;
      wndclass.lpfnWndProc   = WndProc;
      wndclass.cbClsExtra    = 0;
      wndclass.cbWndExtra    = 0;
      wndclass.hInstance     = hInstance;
      wndclass.hIcon         = LoadIcon (NULL,IDI_APPLICATION);
      wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW);
      wndclass.hbrBackground = (HBRUSH) (COLOR_WINDOW + 1);
      wndclass.lpszMenuName  = szAppName;
      wndclass.lpszClassName = szAppName;
      if (!RegisterClass (&wndclass))
         {
         MessageBox (NULL, "RegisterClass failure", szAppName, MB_OK);
         return 0;
```

```
         }
      }
   hWnd = CreateWindow (szAppName, szAppName, WS_OVERLAPPEDWINDOW |
      WS_CLIPCHILDREN, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,
      CW_USEDEFAULT, NULL, NULL, hInstance, NULL);
   if (hWnd == NULL)
      {
      MessageBox (NULL, "CreateWindow failure", szAppName, MB_OK);
      return 0;
      }
// Show the main window
   ShowWindow (hWnd, nCmdShow);
   UpdateWindow (hWnd);
// Play the movies
   while (GetMessage (&msg, NULL, 0, 0))
      {
      TranslateMessage (&msg);
      DispatchMessage (&msg);
      }
// Cut the connections to QuickTime for Windows
   ExitMovies ();
   QTTerminate ();
// Return to Windows
   return msg.wParam;
   }

long FAR PASCAL __export WndProc (HWND hWnd, UINT message,
   WPARAM wParam, LPARAM lParam)
   {
   OPENFILENAME ofn;
   PAINTSTRUCT ps;
   BOOL bLeft;
   POINT ptMovie;
   MovieFile mfMovie;
   static Movie mLeft, mRight;
   static char szDirName [256];
   static char szFile [256];
   static char szFileTitle [256];

// Drive the movie controllers
   if (MCIsPlayerMessage (mcLeft, hWnd, message, wParam, lParam)
    || MCIsPlayerMessage (mcRight, hWnd, message, wParam, lParam))
      return 0;
// Process window messages
   switch (message)
      {
   // Create empty movie controllers when main window is created
      case WM_CREATE:
         SetRectEmpty (&rcMovieBox);
         SetRectEmpty (&rcClient);
         mcLeft = NewMovieController (NULL, &rcClient,
            mcNotVisible, hWnd);
         mcRight = NewMovieController (NULL, &rcClient,
            mcNotVisible, hWnd);
         return 0;
   // Process menu commands
      case WM_COMMAND:
```

```
switch (wParam)
   {
// Use COMMDLG to open a movie file
   case IDM_OPEN:
      memset (&ofn, 0, sizeof (ofn));
      ofn.lStructSize = sizeof (ofn);
      ofn.hwndOwner = hWnd;
      ofn.lpstrFilter = "Movies (*.mov)\0*.mov\0\0";
      ofn.nFilterIndex = 1;
      ofn.lpstrFile = szFile;
      ofn.nMaxFile = sizeof (szFile);
      ofn.lpstrFileTitle = szFileTitle;
      ofn.nMaxFileTitle = sizeof (szFileTitle);
      ofn.lpstrInitialDir =
         _getcwd (szDirName, sizeof (szDirName));
      ofn.Flags = OFN_PATHMUSTEXIST | OFN_FILEMUSTEXIST;
      if (GetOpenFileName (&ofn) &&
         (OpenMovieFile (ofn.lpstrFile, &mfMovie,
         OF_READ) == noErr))
         {
         RECT rcGrowBox;
      // Dispose of any existing movies
         DisposeMovie (mLeft);
         DisposeMovie (mRight);
      // Extract two instances of the same movie
         NewMovieFromFile (&mLeft, mfMovie, NULL, NULL,
            0, NULL);
         NewMovieFromFile (&mRight, mfMovie, NULL, NULL,
            0, NULL);
         CloseMovieFile (mfMovie);
      // Get the normal dimensions of the movie
         GetMovieBox (mLeft, &rcMovieBox);
         OffsetRect (&rcMovieBox, -rcMovieBox.left,
            -rcMovieBox.top);
      // Calculate initial positions of controllers
         GetClientRect (hWnd, &rcClient);
         rcLeft.top = rcRight.top = rcClient.top +
           (rcClient.bottom / 2) - (rcMovieBox.bottom / 2);
         rcLeft.bottom = rcRight.bottom = rcClient.top +
           (rcClient.bottom / 2) + (rcMovieBox.bottom / 2);
         rcLeft.left = (rcClient.right / 4)
           - (rcMovieBox.right / 2);
         rcLeft.right = rcLeft.left + rcMovieBox.right;
         rcRight.left = (rcClient.right / 2)
           + (rcClient.right / 4)
           - (rcMovieBox.right / 2);
         rcRight.right = rcRight.left + rcMovieBox.right;
      // Associate the movies with the existing controllers
         ptMovie.x = rcLeft.left;
         ptMovie.y = rcLeft.top;
         MCSetMovie (mcLeft, mLeft, hWnd, ptMovie);
         ptMovie.x = rcRight.left;
         ptMovie.y = rcRight.top;
         MCSetMovie (mcRight, mRight, hWnd, ptMovie);
      // Pause the movies
         MCDoAction (mcLeft, mcActionPlay, 0);
         MCDoAction (mcRight, mcActionPlay, 0);
```

```
            // Center the movies
               MCPositionController (mcLeft, &rcLeft,
                  NULL, mcTopLeftMovie + mcScaleMovieToFit);
               MCPositionController (mcRight, &rcRight,
                  NULL, mcTopLeftMovie + mcScaleMovieToFit);
            // Make the controllers visible
               MCSetVisible (mcLeft, TRUE);
               MCSetVisible (mcRight, TRUE);
            // Make both movies active and the right mc inactive
               SetMovieActive (mLeft, TRUE);
               SetMovieActive (mRight, TRUE);
               MCActivate (mcRight, hWnd, FALSE);
            // Eliminate the grow boxes
               SetRectEmpty (&rcGrowBox);
               MCDoAction (mcLeft, mcActionSetGrowBoxBounds,
                  &rcGrowBox);
               MCDoAction (mcRight, mcActionSetGrowBoxBounds,
                  &rcGrowBox);
               }
            return 0;
// Change active controller to attached
      case IDM_ATTACH:
         MCSetControllerAttached (mcActive, TRUE);
         return 0;
// Change active controller to detached
      case IDM_DETACH:
         {
         RECT rcMCRect;
         SHORT sMCHeight;
      // Detach the controller
         MCSetControllerAttached (mcActive, FALSE);
      // Choose the appropriate movie/movie controller
         if (mcActive == mcLeft)
            {
         // Get the bounds rect for the controller only
         // since it is now detached
            MCGetControllerBoundsRect (mcLeft, &rcMCRect);
            OffsetRect (&rcMCRect, -rcMCRect.left, -rcMCRect.top);
         // Save the controller height
            sMCHeight = rcMCRect.bottom - rcMCRect.top;
         // Move the controller down
            memcpy (&rcMCRect, &rcLeft, sizeof (RECT));
            rcMCRect.top = rcLeft.bottom +
               (rcMovieBox.bottom / 2);
            rcMCRect.bottom = rcMCRect.top + sMCHeight;
            MCPositionController (mcLeft, &rcLeft, &rcMCRect,
               mcTopLeftMovie);
            }
         else
            {
         // Get the bounds rect for the controller only
         // since it is now detached
            MCGetControllerBoundsRect (mcRight, &rcMCRect);
            OffsetRect (&rcMCRect, -rcMCRect.left, -rcMCRect.top);
         // Save the controller height
            sMCHeight = rcMCRect.bottom - rcMCRect.top;
         // Move the controller down
```

```
                memcpy (&rcMCRect, &rcRight, sizeof (RECT));
                rcMCRect.top = rcRight.bottom +
                   (rcMovieBox.bottom / 2);
                rcMCRect.bottom = rcMCRect.top + sMCHeight;
                MCPositionController (mcRight, &rcRight, &rcMCRect,
                   mcTopLeftMovie);
                }
             }
             return 0;
          }
          return 0;
// Center the controllers in the left and right halves of the window
   case WM_SIZE:
   // Attach the controllers
      MCSetControllerAttached (mcLeft, TRUE);
      MCSetControllerAttached (mcRight, TRUE);
      CalcSize (hWnd);
      MCSetControllerBoundsRect (mcLeft, &rcLeft);
      MCSetControllerBoundsRect (mcRight, &rcRight);
      return 0;
   case WM_LBUTTONDOWN:
      {
      SFIXED sfxVolume;
   // Activate the controller selected by the mouse click
      GetClientRect (hWnd, &rcClient);
      bLeft = (SHORT) (LOWORD (lParam)) < ((rcClient.right -
         rcClient.left) / 2);
      mcActive = bLeft ? mcLeft : mcRight;
      MCActivate (mcLeft, hWnd, bLeft);
      MCActivate (mcRight, hWnd, !bLeft);
   // Disable sound and keyboard interface for appropriate controller
      if (mcActive == mcLeft)
         {
         MCDoAction (mcRight, mcActionGetVolume, (LPVOID)
            &sfxVolume);
         sfxVolume = - (abs (sfxVolume));
         MCDoAction (mcRight, mcActionSetVolume, (LPVOID) sfxVolume);
         MCDoAction (mcRight, mcActionSetKeysEnabled,
            (LPVOID) FALSE);
         }
      else
         {
         MCDoAction (mcLeft, mcActionGetVolume, (LPVOID) &sfxVolume);
         sfxVolume = - (abs (sfxVolume));
         MCDoAction (mcLeft, mcActionSetVolume, (LPVOID) sfxVolume);
         MCDoAction (mcLeft, mcActionSetKeysEnabled, (LPVOID) FALSE);
         }
   // Enable sound and keyboard for active controller
      MCDoAction (mcActive, mcActionGetVolume, (LPVOID) &sfxVolume);
      sfxVolume = abs (sfxVolume);
      MCDoAction (mcActive, mcActionSetVolume, (LPVOID) sfxVolume);
      MCDoAction (mcActive, mcActionSetKeysEnabled, (LPVOID) TRUE);
      }
      return 0;
// Repaint the Window
   case WM_PAINT:
      if (!BeginPaint (hWnd, &ps))
```

```
            return 0;
         EndPaint (hWnd, &ps);
         return 0;
   // Destroy the movies and controllers when the window is destroyed
      case WM_DESTROY:
         DisposeMovieController (mcLeft);
         DisposeMovieController (mcRight);
         DisposeMovie (mLeft);
         DisposeMovie (mRight);
         PostQuitMessage (0);
         return 0;
      }
// Return to Windows
   return DefWindowProc (hWnd, message, wParam, lParam);
   }

VOID CalcSize (HWND hWndCaller)
   {
   RECT rcBounds;

   GetClientRect (hWndCaller, &rcClient);
   MCGetControllerBoundsRect (mcLeft, &rcBounds);
   OffsetRect (&rcBounds, -rcBounds.left, -rcBounds.top);
   rcLeft.top = rcRight.top = rcClient.top +
      (rcClient.bottom / 2) - (rcBounds.bottom / 2);
   rcLeft.bottom = rcRight.bottom = rcClient.top +
      (rcClient.bottom / 2) + (rcBounds.bottom / 2);
   rcLeft.left = (rcClient.right / 4) - (rcBounds.right / 2);
   rcLeft.right = (rcClient.right / 4) + (rcBounds.right / 2);
   MCGetControllerBoundsRect (mcRight, &rcBounds);
   OffsetRect (&rcBounds, -rcBounds.left, -rcBounds.top);
   rcRight.left = (rcClient.right / 2) + (rcClient.right / 4)
      - (rcBounds.right / 2);
   rcRight.right = (rcClient.right / 2) + (rcClient.right / 4)
      + (rcBounds.right / 2);
   }
```

## 3. Understanding Active and Inactive Movie States

As we learned in the overview, both movies and movie controllers have active and inactive states. While they are easy to set, it is still important to remember two things: these states do not affect QuickTime for Windows programs in parallel ways, and more than one movie or controller can be active simultaneously.

A movie's state can be set by SetMovieActive, whose parameters are the movie object and a value of either `TRUE` (for active) or `FALSE` (for inactive). An inactive movie simply is not played--it does not receive cycles from QuickTime for Windows' internal scheduler. Don't confuse a movie's active state with its playing/paused state. In other words, calling SetMovieActive should not be used to start or stop playing a movie.

A movie controller's state can be set by MCActivate with its last parameter set to `TRUE` or `FALSE`. Again, since movie controllers generally mirror the behavior of standard Windows controls, it is useful to view an inactive movie controller as a disabled Windows control. It cannot receive user input (i.e. mouse clicks, since keyboard input is enabled separately) and its appearance is grayed. Movie controllers are created with an active state by default.

A movie/movie controller pair can easily have opposing states. For instance, an active movie can have an inactive controller, and vice versa. In the former case, a playing movie's controller can be deactivated, graying it and prohibiting further user input, but the movie will keep playing. In the latter, clicking the start button on an inactive movie's active controller will not play the movie.

Since more than one movie or movie controller can have active or inactive status under QuickTime for Windows itself, it is the application's responsibility to identify and keep track of its own *application specific* active movies, movie controllers and controller attributes (e.g., sound and keyboard states). Any serious QuickTime for Windows program design must be aware of and incorporate this paradigm if it expects to effectively route events and call QuickTime for Windows functions with appropriate movie and movie controller objects.

STEREO addresses the issue in an elementary way using a variable called `mcActive`. Whenever a movie controller is activated by a user input event (i.e. a mouse click), the movie controller object linked to the window area which received the click is copied into this variable. (This is merely a convention used to simplify our sample program--see the code fragment below). As a result, routines using the program's active movie controller object pass `mcActive` instead of the variable that received the original controller object.

STEREO calls MCActivate on what it deems *its* non-active controller with the last parameter set to `FALSE`, setting it to a QuickTime for Windows inactive state. This in turn causes the controller's elements to be grayed (see Figure 21, below).

```
case WM_LBUTTONDOWN:
   •
   •
// Activate the controller selected by the mouse click
   GetClientRect (hWnd, &rcClient);
   bLeft = (SHORT) (LOWORD (lParam)) < ((rcClient.right -
      rcClient.left) / 2);
   mcActive = bLeft ? mcLeft : mcRight;
   MCActivate (mcLeft, hWnd, bLeft);
   MCActivate (mcRight, hWnd, !bLeft);
   •
   •
   return 0;
```

## 4. Visualizing Attached and Detached Movie Controllers

A movie controller is attached to or detached from a movie also by an explicit QuickTime for Windows function call, such as MCSetControllerAttached. Once attached, it is automatically associated and normally appears joined to the bottom edge of the movie (under uncommon circumstances they may be programatically attached but not physically joined). When the controller is used for resizing, both it and the movie grow or shrink together. If the application repositions either one of them, they both travel in unison.

Detached movie controllers are not joined physically to their movies (as above, this is the normal condition--sometimes they may be programatically detached but not separated). Although they play their movies just like attached controllers, repositioning or resizing one does not necessarily affect the other. As you will see, detached movie controllers can perform some very useful functions.

You cannot create a detached movie controller from scratch. If your program requires one, you have to detach an existing attached controller. STEREO plays with this idea a little by creating a pair of movie controllers using NewMovieController with its first parameter set to NULL, then associating them with movies when they are opened.

The other parameters are the address of the `RECT` containing the controller's screen coordinates--in this case all zeros, the controller creation flags and the parent window handle.

STEREO's two movie controllers are created early (and invisibly) to simplify the flow of this tutorial application. Not only that, they also play the same movie--eventually. Nevertheless, the program demonstrates several important differences between attached and detached controllers, as well as QuickTime for Windows' high degree of flexibility in handling them and its other components.

## 5. Attaching Movie Controllers to Movies

As explained in the overview, the function MCNewAttachedController is often used to both associate and attach movies and movie controllers. STEREO uses MCSetMovie instead to simply associate them. Its significant parameters are `PtLeft` and `PtRight`, the upper left corners of the movies relative to their parent window.

STEREO calls MCSetMovie on its existing controllers as soon as a movie is selected for opening, detaches them for proper sizing of their movie rectangles, then re-attaches them and makes them visible. We now have two otherwise normal movies with attached movie controllers ready for playing. But this is not the only way to attach a movie controller to a movie, as you can infer by using the **Action** menu to dynamically detach and re-attach them even while they are running.

## 6. Detaching and Re-attaching a Movie Controller

Pulling down the **Action** menu gives you **Attach Controller** and **Detach Controller** options for the application's active movie controller. If the controller is not attached, selecting **Attach Controller** causes it to jump to its appropriate attached position. The routine used for this purpose is MCSetControllerAttached, which takes as parameters the movie controller object and the boolean value `TRUE`.

Selecting the **Detach Controller** menu item when the controller is currently attached to a movie triggers two significant events. First, MCSetControllerAttached is called with a value of `FALSE`. This alone, however, does not physically separate the movie controller from the movie. To split them apart you need MCPositionController.

The parameters of interest are the addresses of the `RECT` structures for the desired coordinates of the movie and the movie controller. If we had wanted to *query* the attachment state of the movie controller so we could, say, gray the appropriate menu item, we could have used the routine MCIsControllerAttached.

STEREO uses numbers which set the resulting detached controllers at arbitrary distances slightly below their movies, but your future programs could use values which have real meaning in developing a consistent user interface for your QuickTime for Windows applications. For example, your detached movie controllers could be handled like custom menus or tool bars in terms of their default positions and where the user of the application might expect to find them if not attached to their movies.

## 7. Resizing Movies and Movie Controllers

Just as it is the application's job to designate and track its own active movie controller(s), it must also handle changing movie and movie controller dimensions if the application's window is resized. STEREO does this under the WM_SIZE case in its window procedure, using the routine MCSetControllerBoundsRect.

When a WM_SIZE message is received, the program gets the coordinates of the client rectangle. It then bisects that area vertically to derive left and right sub-rectangles for each movie, which are supplied with slight offsets to MCSetControllerBoundsRect The function centers the resized movies and controllers in the new rectangles.

In your own QuickTime for Windows programs you may not want to resize your movies with your program windows. STEREO does it to show you the power of this particular call.

## 8. Calling MCIsPlayerMessage More than Once

Each movie controller that you want to receive messages must have a corresponding MCIsPlayerMessage call in the window procedure of its parent window. STEREO.C contains two instances of the routine, each one with a different controller object.

As your QuickTime for Windows programs get more complex, this is one of the points where you should carefully design the handling of their movie controller messages. For instance, you might keep an array of controller objects and call MCIsPlayerMessage in a `for` loop, passing specific objects conditionally, etc. Again, you will have to decide the best way to handle this.

## 9. Running STEREO.EXE

When STEREO.EXE is executed, the movie controllers will not be visible in the client area of the main window, since no movie is open yet. When a movie is opened from the file menu, each controller will become visible and attach itself to one of the two movies which will appear in STEREO's client area. The left one is initially set to an active state, and the right one made inactive.

As you experiment with the **Action** menu, your movie controllers will become detached and re-attached. You will notice that while the visual parts of both movies can play simultaneously, only the sound track of the active movie will be played. This is a Windows limitation--not a condition that can be controlled with the QuickTime for Windows API.

# D. BIGEIGHT - Movie Controller Attributes

# 1. Introduction

Beyond overt characteristics like association and attachment, movie controllers have many other useful attributes. The next sample program, BIGEIGHT.EXE, allows you to switch on and off eight of these attributes for a single detached movie controller. The attributes demonstrated are: controller visibility, speaker button visibility, step button visibility, grow box visibility, sound availability, keyboard interface availability, movie looping and palindrome looping modes.

## 2. The BIGEIGHT Source Code

BIGEIGHT.MAK

```
ALL : BIGEIGHT.EXE

BIGEIGHT.OBJ : BIGEIGHT.C
 cl -c -AS -DSTRICT -G2 -Zpel -W3 -WX -Od bigeight.c

BIGEIGHT.RES : BIGEIGHT.RC BIGEIGHT.H
 rc -r bigeight.rc

BIGEIGHT.EXE : BIGEIGHT.OBJ BIGEIGHT.RES BIGEIGHT.DEF
 link /nod /a:16 bigeight, bigeight.exe, nul, qtw libw slibcew, \
    bigeight.def
 rc bigeight.res
```

BIGEIGHT.DEF

```
NAME        BIGEIGHT
DESCRIPTION 'Sample Application'
EXETYPE     WINDOWS
STUB        'winstub.exe'
CODE        PRELOAD MOVEABLE DISCARDABLE
DATA        PRELOAD MOVEABLE MULTIPLE
HEAPSIZE    1024
STACKSIZE   16384
```

BIGEIGHT.H

```
#define IDM_CONTROLLER     1
#define IDM_GROW_BOX       2
#define IDM_KEYBOARD       3
#define IDM_LOOPING        4
#define IDM_PALINDROME     5
#define IDM_SOUND          6
#define IDM_SPEAKER_BUTTON 7
#define IDM_STEP_BUTTONS   8
```

BIGEIGHT.RC

```
#include <windows.h>
#include "bigeight.h"

bigeight MENU
  {
  POPUP "&Attributes"
    {
    MENUITEM "&Hide Controller", IDM_CONTROLLER
    MENUITEM "&Hide Step Buttons", IDM_STEP_BUTTONS
    MENUITEM "&Hide Speaker Button", IDM_SPEAKER_BUTTON
    MENUITEM "&Hide Grow Box", IDM_GROW_BOX
    MENUITEM SEPARATOR
```

```
    MENUITEM "&Disable Keyboard Interface", IDM_KEYBOARD
    MENUITEM "&Disable Sound", IDM_SOUND
    MENUITEM "&Enable Looping", IDM_LOOPING
    MENUITEM "&Enable Palindrome Looping", IDM_PALINDROME
    }
  }
```

BIGEIGHT.C

```
#include <windows.h>
#include <qtw.h>
#include "bigeight.h"

long FAR PASCAL __export WndProc (HWND, UINT, WPARAM, LPARAM);

MovieController mcController;

int PASCAL WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
   LPSTR lpszCmdParam, int nCmdShow)
   {
   static char szAppName[] = "BigEight";
   HWND        hWnd;
   MSG         msg;
   WNDCLASS    wndclass;
   Movie       mMovie;
   RECT        rcMovie, rcMovieBox;
   MovieFile   mfMovie;

// Establish links to QuickTime for Windows
   if (QTInitialize (NULL))
      {
      MessageBox (NULL, "QTInitialize failure", szAppName, MB_OK);
      return 0;
      }
// Allocate memory required for playing movies
   if (EnterMovies ())
      {
      MessageBox (NULL, "EnterMovies failure", szAppName, MB_OK);
      return 0;
      }
// Register and create main window
   if (!hPrevInstance)
      {
      wndclass.style         = CS_DBLCLKS | CS_HREDRAW | CS_VREDRAW;
      wndclass.lpfnWndProc   = WndProc;
      wndclass.cbClsExtra    = 0;
      wndclass.cbWndExtra    = 0;
      wndclass.hInstance     = hInstance;
      wndclass.hIcon         = LoadIcon (NULL,IDI_APPLICATION);
      wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW);
      wndclass.hbrBackground = (HBRUSH) (COLOR_WINDOW + 1);
      wndclass.lpszMenuName  = szAppName;
      wndclass.lpszClassName = szAppName;
      if (!RegisterClass (&wndclass))
         {
         MessageBox (NULL, "RegisterClass failure", szAppName, MB_OK);
```

```
            return 0;
            }
         }
      hWnd = CreateWindow(szAppName, szAppName, WS_OVERLAPPEDWINDOW |
         WS_CLIPCHILDREN, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,
         CW_USEDEFAULT, NULL, NULL, hInstance, NULL);
      if (hWnd == NULL)
         {
         MessageBox (NULL, "CreateWindow failure", szAppName, MB_OK);
         return 0;
         }
// Instantiate the movie
      if (OpenMovieFile ("SAMPLE.MOV", &mfMovie, OF_READ) != noErr)
         {
         MessageBox (NULL, "OpenMovieFile failure", szAppName, MB_OK);
         return 0;
         }
      NewMovieFromFile (&mMovie, mfMovie, NULL, NULL, 0, NULL);
      CloseMovieFile (mfMovie);
// Instantiate the movie controller
      GetMovieBox (mMovie, &rcMovieBox);
      OffsetRect(&rcMovieBox, -rcMovieBox.left, -rcMovieBox.top);
      GetClientRect (hWnd, &rcMovie);
      rcMovie.top = (rcMovie.bottom / 2) - (rcMovieBox.bottom / 2);
      rcMovie.bottom = rcMovie.top + rcMovieBox.bottom;
      rcMovie.left = (rcMovie.right / 2) - (rcMovieBox.right / 2);
      rcMovie.right = rcMovie.left + rcMovieBox.right;
      mcController = NewMovieController (mMovie, &rcMovie,
         mcTopLeftMovie + mcScaleMovieToFit, hWnd);
// Make the movie paused initially
      MCDoAction (mcController, mcActionPlay, 0);
// Enable the keyboard interface
      MCDoAction (mcController, mcActionSetKeysEnabled, (LPVOID) TRUE);
// Make the movie active
      SetMovieActive (mMovie, TRUE);
// Make the main window visible
      ShowWindow (hWnd, nCmdShow);
      UpdateWindow (hWnd);
// Play the movie
      while (GetMessage (&msg, NULL, 0, 0))
         {
         TranslateMessage (&msg);
         DispatchMessage (&msg);
         }
// Destroy the movie controller
      DisposeMovieController (mcController);
// Destroy the movie
      DisposeMovie (mMovie);
// Cut the connections to QuickTime for Windows
      ExitMovies ();
      QTTerminate ();
// Return to Windows
      return msg.wParam;
      }

long FAR PASCAL __export WndProc (HWND hWnd, UINT message,
      WPARAM wParam, LPARAM lParam)
```

```
    {
    PAINTSTRUCT ps;
    RECT        rcGrowBox;

    static BOOL bControllerVisible = TRUE;
    static BOOL bGrowBoxVisible = TRUE;
    static BOOL bKeysEnabled = TRUE;
    static BOOL bLoopingEnabled = FALSE;
    static BOOL bPalindromeEnabled = FALSE;
    static BOOL bSoundEnabled = TRUE;
    static BOOL bSpeakerVisible = TRUE;
    static BOOL bSteppersVisible = TRUE;

// Drive the movie controller
    if (MCIsPlayerMessage (mcController, hWnd, message, wParam, lParam))
       return 0;
// Process the windows message
    switch (message)
       {
       case WM_COMMAND:
          {
          HANDLE hMenu;
          hMenu= GetMenu (hWnd);
          switch (wParam)
             {
             case IDM_CONTROLLER:
                {
                if (bControllerVisible == FALSE)
                   {
                // Change the controller menu item
                   ModifyMenu (hMenu, IDM_CONTROLLER, MF_BYCOMMAND |
                      MF_STRING, IDM_CONTROLLER,
                      (LPSTR) "Hide Controller");
                   bControllerVisible = TRUE;
                // Show the controller
                   MCSetVisible (mcController, TRUE);
                // Ungray the other menu itmes
                   EnableMenuItem (hMenu, IDM_STEP_BUTTONS, MF_ENABLED);
                   EnableMenuItem (hMenu, IDM_SPEAKER_BUTTON,
                      MF_ENABLED);
                   EnableMenuItem (hMenu, IDM_GROW_BOX, MF_ENABLED);
                   EnableMenuItem (hMenu, IDM_KEYBOARD, MF_ENABLED);
                   EnableMenuItem (hMenu, IDM_SOUND, MF_ENABLED);
                   EnableMenuItem (hMenu, IDM_LOOPING, MF_ENABLED);
                   EnableMenuItem (hMenu, IDM_PALINDROME, MF_ENABLED);
                   }
                else
                   {
                // Change the controller menu item
                   ModifyMenu (hMenu, IDM_CONTROLLER, MF_BYCOMMAND |
                      MF_STRING, IDM_CONTROLLER,
                      (LPSTR) "Show Controller");
                   bControllerVisible = FALSE;
                // Hide the controller
                   MCSetVisible (mcController, FALSE);
                // Grey the rest of the menu items
                   EnableMenuItem (hMenu, IDM_STEP_BUTTONS, MF_GRAYED);
```

```
            EnableMenuItem (hMenu, IDM_SPEAKER_BUTTON, MF_GRAYED);
            EnableMenuItem (hMenu, IDM_GROW_BOX, MF_GRAYED);
            EnableMenuItem (hMenu, IDM_KEYBOARD, MF_GRAYED);
            EnableMenuItem (hMenu, IDM_SOUND, MF_GRAYED);
            EnableMenuItem (hMenu, IDM_LOOPING, MF_GRAYED);
            EnableMenuItem (hMenu, IDM_PALINDROME, MF_GRAYED);
            }
        }
        break;
    case IDM_STEP_BUTTONS:
        {
        LONG lFlags;
        if (bSteppersVisible == FALSE)
            {
        // Change the step button menu item
            ModifyMenu (hMenu, IDM_STEP_BUTTONS, MF_BYCOMMAND |
                MF_STRING, IDM_STEP_BUTTONS,
                (LPSTR) "Hide Step Buttons");
            bSteppersVisible = TRUE;
        // Restore the step buttons
            MCDoAction (mcController, mcActionGetFlags, &lFlags);
            lFlags &= ~mcFlagSuppressStepButtons;
            MCDoAction (mcController, mcActionSetFlags,
                (LPVOID) lFlags);
            }
        else
            {
        // Change the step button menu item
            ModifyMenu (hMenu, IDM_STEP_BUTTONS, MF_BYCOMMAND |
                MF_STRING, IDM_STEP_BUTTONS,
                (LPSTR) "Show Step Buttons");
            bSteppersVisible = FALSE;
        // Hide the step buttons
            MCDoAction (mcController, mcActionGetFlags, &lFlags);
            lFlags |= mcFlagSuppressStepButtons;
            MCDoAction (mcController, mcActionSetFlags,
                (LPVOID) lFlags);
            }
        }
        break;
    case IDM_SPEAKER_BUTTON:
        {
        LONG lFlags;
        if (bSpeakerVisible == FALSE)
            {
        // Change the speaker button menu item
            ModifyMenu (hMenu, IDM_SPEAKER_BUTTON, MF_BYCOMMAND |
                MF_STRING, IDM_SPEAKER_BUTTON,
                (LPSTR) "Hide Speaker Button");
            bSpeakerVisible = TRUE;
        // Restore the speaker button
            MCDoAction (mcController, mcActionGetFlags, &lFlags);
            lFlags &= ~mcFlagSuppressSpeakerButton;
            MCDoAction (mcController, mcActionSetFlags,
                (LPVOID) lFlags);
            }
        else
```

```
      {
   // Change the speaker button menu item
      ModifyMenu (hMenu, IDM_SPEAKER_BUTTON, MF_BYCOMMAND |
         MF_STRING, IDM_SPEAKER_BUTTON,
         (LPSTR) "Show Speaker Button");
      bSpeakerVisible = FALSE;
   // Hide the speaker button
      MCDoAction (mcController, mcActionGetFlags, &lFlags);
      lFlags |= mcFlagSuppressSpeakerButton;
      MCDoAction (mcController, mcActionSetFlags,
         (LPVOID) lFlags);
      }
   }
   break;
case IDM_GROW_BOX:
   {
   if (bGrowBoxVisible == FALSE)
      {
   // Change the grow box menu item
      ModifyMenu (hMenu, IDM_GROW_BOX, MF_BYCOMMAND |
         MF_STRING, IDM_GROW_BOX, (LPSTR) "Hide Grow Box");
      bGrowBoxVisible = TRUE;
   // Set the grow box bounds to make it visible
      GetClientRect (hWnd, &rcGrowBox);
      MCDoAction (mcController, mcActionSetGrowBoxBounds,
         &rcGrowBox);
      }
   else
      {
   // Change the grow box menu item
      ModifyMenu (hMenu, IDM_GROW_BOX, MF_BYCOMMAND |
         MF_STRING, IDM_GROW_BOX,(LPSTR) "Show Grow Box");
      bGrowBoxVisible = FALSE;
   // Set the grow box bounds to all zeros to hide it
      SetRectEmpty (&rcGrowBox);
      MCDoAction (mcController, mcActionSetGrowBoxBounds,
         &rcGrowBox);
      }
   }
   break;
case IDM_KEYBOARD:
   {
   if (bKeysEnabled == FALSE)
      {
   // Change the keyboard interface menu item
      ModifyMenu (hMenu, IDM_KEYBOARD, MF_BYCOMMAND |
         MF_STRING, IDM_KEYBOARD,
         (LPSTR) "Disable Keyboard Interface");
      bKeysEnabled = TRUE;
   // Enable the keyboard interface
      MCDoAction (mcController, mcActionSetKeysEnabled,
         (LPVOID) TRUE);
      }
   else
      {
   // Change the keyboard interface menu item
      ModifyMenu (hMenu, IDM_KEYBOARD, MF_BYCOMMAND |
```

```
            MF_STRING, IDM_KEYBOARD,
            (LPSTR) "Enable Keyboard Interface");
         bKeysEnabled = FALSE;
      // Disable the keyboard interface
         MCDoAction (mcController, mcActionSetKeysEnabled,
            (LPVOID) FALSE);
         }
      }
      break;
   case IDM_SOUND:
      {
      SFIXED sfxVolume;
      if (bSoundEnabled == FALSE)
         {
      // Change the sound menu item
         ModifyMenu (hMenu, IDM_SOUND, MF_BYCOMMAND |
            MF_STRING, IDM_SOUND, (LPSTR) "Disable Sound");
      // Restore the sound
         MCDoAction (mcController, mcActionGetVolume,
            (LPVOID) &sfxVolume);
         sfxVolume = abs (sfxVolume);
         MCDoAction (mcController, mcActionSetVolume,
            (LPVOID) sfxVolume);
         bSoundEnabled = TRUE;
         }
      else
         {
      // Mute the sound
         MCDoAction (mcController, mcActionGetVolume,
            (LPVOID) &sfxVolume);
         sfxVolume = -(abs (sfxVolume));
         MCDoAction (mcController, mcActionSetVolume,
            (LPVOID) sfxVolume);
         bSoundEnabled = FALSE;
         }
      }
      break;
   case IDM_LOOPING:
      {
      if (bLoopingEnabled == FALSE)
         {
      // Change the looping menu item
         ModifyMenu (hMenu, IDM_LOOPING, MF_BYCOMMAND |
            MF_STRING, IDM_LOOPING, (LPSTR) "Disable Looping");
         bLoopingEnabled = TRUE;
      // Enable looping
         MCDoAction (mcController, mcActionSetLooping,
            (LPVOID) TRUE);
         }
      else
         {
      // Change the looping menu item
         ModifyMenu (hMenu, IDM_LOOPING, MF_BYCOMMAND |
            MF_STRING, IDM_LOOPING, (LPSTR) "Enable Looping");
         bLoopingEnabled = FALSE;
      // Disable looping
         MCDoAction (mcController, mcActionSetLooping,
```

```
                    (LPVOID) FALSE);
                }
            }
            break;
         case IDM_PALINDROME:
            {
            if (bPalindromeEnabled == FALSE)
               {
            // Change the palindrome menu item
               ModifyMenu (hMenu, IDM_PALINDROME, MF_BYCOMMAND |
                  MF_STRING, IDM_PALINDROME,
                  (LPSTR) "Disable Palindrome Looping");
               bPalindromeEnabled = TRUE;
            // Enable palindrome looping
               MCDoAction (mcController, mcActionSetLooping,
                  (LPVOID) TRUE);
               MCDoAction (mcController, mcActionSetLoopIsPalindrome,
                  (LPVOID) TRUE);
               }
            else
               {
            // Change the palindrome menu item
               ModifyMenu (hMenu, IDM_PALINDROME, MF_BYCOMMAND |
                  MF_STRING, IDM_PALINDROME,
                  (LPSTR) "Enable Palindrome Looping");
               bPalindromeEnabled = FALSE;
            // Disable palindrome looping
               MCDoAction (mcController, mcActionSetLooping,
                  (LPVOID) FALSE);
               MCDoAction (mcController, mcActionSetLoopIsPalindrome,
                  (LPVOID) FALSE);
               }
            }
            break;
         }
      }
      return 0;
   case WM_PAINT:
      if (!BeginPaint (hWnd, &ps))
         return 0;
      EndPaint (hWnd, &ps);
      return 0;
   case WM_DESTROY:
      PostQuitMessage (0);
      return 0;
   }
// Return to Windows
   return DefWindowProc (hWnd, message, wParam, lParam);
   }
```

## 3. The Power of MCDoAction

One of the most powerful routines in the QuickTime for Windows API is MCDoAction. As you can see in the BIGEIGHT.C listing, this function is used to change and query Movie Controller attributes. In QuickTime for Windows' grand scheme, however, MCDoAction is a cornerstone routine which can be used to dictate most of the Movie Controller's behavior. It is so versatile, in fact, that several other QuickTime for Windows routines use it internally to accomplish their particular tasks.

MCDoAction works by taking as its second parameter a particular defined action. There are approximately thirty-five such *mcActions* in the QuickTime for Windows API, ranging from starting the movie to toggling low-level attributes. In most cases, a third parameter is required to modify the task of the *mcAction* parameter. Often this is a boolean value which turns a certain attribute on or off, or a pointer to a value holding state information:

```
MovieController mcController;
•
•
MCDoAction (mcController, mcActionSetKeysEnabled, (LPVOID) FALSE);
```

# 4. Actions and Flags

There are four components to the methods you use to determine attributes for a movie controller. The first is the collection of *mcActions* used by MCDoAction. A full listing of these actions is provided in the entry for the MCDoAction function.

Second is a group of flags used specifically by MCDoAction when it specifies the *mcActions* mcActionSetFlags or mcActionGetFlags:

| Flag | Function |
|---|---|
| mcFlagSuppressStepButtons | Inhibit display of step buttons |
| mcFlagSuppressSpeakerButton | Inhibit display of speaker button |
| mcFlagsUseWindowPalette | Use a Windows palette to display movies |

BIGEIGHT uses the first and second flags in the above list when it hides its movie controller's step and speaker buttons:

```
case IDM_SPEAKER_BUTTON:
   •
   •
   MCDoAction (mcController, mcActionGetFlags, &lFlags);
   lFlags |= mcFlagSuppressSpeakerButton;
   MCDoAction (mcController, mcActionSetFlags, (LPVOID) lFlags);
```

Use of the flag `mcFlagsUseWindowPalette` is slightly more complex, as it involves the Windows palette manager. Telling a movie controller to use this flag essentially enables it to construct a custom color palette based on the color values found in the movie.

For instance, a particular movie might be of a sunset with fifty shades of orange. If the normal palette is used, these would all be mapped to a much smaller number of orange-ish hues. If a custom palette is used, additional shades of orange will be available for a much more faithful display. You should note that using `mcFlagsUseWindowPalette` only works with display drivers that support palettes--typically drivers that handle colors at pixel depth eight.

Also be aware that any program you are running that calls `RealizePalette` will distort other visible movies or pictures. This is because the palette on which the other images were based has changed. To restore them as well as possible, it is recommended that each of your QuickTime for Windows applications trap the `WM_PALETTECHANGED` message in its main window procedure. When this message is received, they should repaint their main windows and all child windows (using `InvalidateRect` is recommended) to remap their colors as closely as possible to the newly realized system palette.

The third set of flags constitutes a long integer and can be referred to as the *mcInfoFlags*. These flags hold state information set by MCDoAction with one of its *mcActions*, and can be retrieved by the function MCGetControllerInfo, as we saw in the overview.

The last group of flags are used to set movie controller attributes at creation time, not in conjunction with a MCDoAction call:

| Flag | Function |
|---|---|
| mcTopLeftMovie | positions movie in top left corner of Movie rectangle |
| mcScaleMovieToFit | makes movie fit exactly into movie rectangle |
| mcWithBadge | makes movie controller capable of badge display |
| mcNotVisible | makes movie controller invisible when created |

These self-descriptive flags are used by the routine NewMovieController when a movie controller is created. The first two are used by MCPositionController when a controller is repositioned. BIGEIGHT uses two of them to instantiate its controller:

```
MovieController mcController;
Movie mMovie;
RECT rcMovie;
•
•
mcController = NewMovieController (mMovie, &rcMovie,
   mcTopLeftMovie + mcScaleMovieToFit + mcWithBadge, hWnd);
```

As the states of these flags are not maintained by a movie controller, the QuickTime for Windows API does not provide a way to query them.

## 5. Regulating Movie Controller Attributes with MCDoAction

One of the first uses BIGEIGHT makes of MCDoAction is to enable the movie controller's keyboard interface:

```
MovieController mc;
•
•
MCDoAction (mcController, mcActionSetKeysEnabled, (LPVOID) TRUE);
```

An inactive keyboard interface is the default attribute for a new movie controller, but you can enable it at any time by calling MCDoAction as above with the last parameter set to `TRUE`. BIGEIGHT lets you toggle this attribute on and off using its attributes menu. Since all movie controllers with their keyboard interface turned on receive keystrokes, you will have to manage this attribute for each controller in a multi-movie application.

The default visible attributes of a movie controller are the speaker button, the start/pause button, the slider, the step buttons and the grow box (for attached controllers only). Of these, the speaker, the steppers and the grow box can be made invisible, though not all in the same way.

A controller's speaker and step buttons may be hidden or restored using MCDoAction with mcActionSetFlags and either `mcFlagSuppressSpeakerButton` or `mcFlagSuppressStepButton`, respectively:

```
case IDM_STEP_BUTTONS:
    •
    •
    MCDoAction (mcController, mcActionGetFlags, &lFlags);
    lFlags |= mcFlagSuppressStepButtons;
    MCDoAction (mcController, mcActionSetFlags, (LPVOID) lFlags);
```

In BIGEIGHT, the current flags are retrieved, modified and reset in as short a time as possible. This is good QuickTime for Windows programming style for a couple of reasons. First, you should not attempt to maintain a set of these flags yourself. The are managed by QuickTime for Windows and subject to its own internal functionality. Also, like Windows itself, QuickTime for Windows is a complex message-based entity that expects you to deal efficiently with any state information it makes available to you.

Hiding the grow box also uses MCDoAction, but with a different action parameter, namely mcActionSetGrowBoxBounds:

```
case IDM_GROW_BOX:
    •
    •
    SetRectEmpty (&rcGrowBox);
    MCDoAction (mcController, mcActionSetGrowBoxBounds, &rcGrowBox);
```

What actually hides the grow box are the dimensions of the third parameter, `rcMovie`, which have all been set to 0 by the Windows function `SetRectEmpty`. This is the only way to hide a movie controller's grow box.

BIGEIGHT calls MCDoAction the same way to restore the grow box, but with a non-zeroed rectangle. In this case, the client area of the parent window is used nominally.

The looping and looping palindrome attributes affect how a movie plays once it has been started by its controller. Simple looping specifies that the movie play continuously from start to finish until it is stopped by the user. Palindrome looping causes it to play continuously back and forth. MCDoAction has defined actions for both the looping and palindrome attributes. The third parameter in either case is a boolean, which is used to toggle the attributes on or off. For palindrome looping to work, both normal looping and palindrome looping have to be enabled.

```
case IDM_PALINDROME:
```

```
    MCDoAction (mcController, mcActionSetLooping, (LPVOID) TRUE);
    MCDoAction (mcController, mcActionSetLoopIsPalindrome,
        (LPVOID) TRUE);
```

To query the state of the looping attributes, you can call MCGetControllerInfo and then examine the variable it fills with the attribute flags discussed above.

Turning the sound off involves using MCDoAction to retrieve the volume value, negating it, then using MCDoAction again reset it to the negative value. To turn it back on, we retrieve the value and reset the absolute value of it.

```
case IDM_SOUND:
    {
    SFIXED sfxVolume;
    if (bSoundEnabled == FALSE)
        {
    // Restore the sound
        MCDoAction (mcController, mcActionGetVolume, (LPVOID) &sfxVolume);
        sfxVolume = abs (sfxVolume);
        MCDoAction (mcController, mcActionSetVolume, (LPVOID) sfxVolume);
        bSoundEnabled = TRUE;
        }
    else
        {
    // Mute the sound
        MCDoAction (mcController, mcActionGetVolume, (LPVOID) &sfxVolume);
        sfxVolume = -(abs (sfxVolume));
        MCDoAction (mcController, mcActionSetVolume, (LPVOID) sfxVolume);
        bSoundEnabled = FALSE;
        }
    }
    break;
```

## 6. Using MCSetVisible

Setting the visibility attribute of a movie controller does not require MCDoAction. Rather it uses the function MCSetVisible, which takes the controller object and a `TRUE` or `FALSE` second parameter to either show or hide it:

```
MovieController mcController;
BOOL bState;
•
•
MCSetVisible (mcController, bState);
```

As noted in the overview, you can hide or restore an existing movie controller to view at any time. You can also specify that it be hidden when created (using the controller creation flags discussed earlier), and then later change its visibility attribute by calling MCSetVisible with a value of `TRUE`.

## 7. Badges

When a movie controller is made invisible, a badge can appear on the face of its associated movie to distinguish it from other types of graphic objects. The ability to display a badge is an attribute set at creation time with the controller creation flag `mcWithBadge` or later with MCDoAction. If this attribute is not set, no badge will appear. BIGEIGHT sets the badge attribute when it creates its controller:

```
Movie mMovie;
MovieController mcController;
RECT rcMovie;
•
•
mcController = NewMovieController (mMovie, &rcMovie,
   mcTopLeftMovie + mcScaleMovieToFit + mcWithBadge, hWnd);
```

Clicking on a badge will hide it and display the movie controller, providing that the `mcWithBadge` flag is set.

If you want to manipulate a badge manually, MCDrawBadge is available. Assuming you do not set the `mcWithBadge` flag, you must be prepared to call this function whenever you want the badge to appear. Since playing the movie will automatically write over an existing badge, there is no specific QuickTime for Windows routine to hide a badge. MCDrawBadge does not set the `mcWithBadge` flag.

The second parameter of MCDrawBadge should always be NULL in this version of QuickTime for Windows. The third is the address of a handle to the badge region (a standard Windows `HRGN`) subsequently available to your program. QuickTime for Windows creates a region describing the area in which it drew the badge, and returns that region to you. It is your responsibility to later delete this region.

```
MovieController mcController;
HRGN hrgnBadge;
•
•
MCDrawBadge (mcController, NULL, &hrgnBadge);
```

A badge is a movie controller attribute even though it is a separate visual object. This assertion is supported by the fact that its availability can be set and queried with MCDoAction, and also at controller creation time along with other attributes.

## 8. Running BIGEIGHT.EXE

The first thing you see when you run BIGEIGHT is a movie positioned near the center of its client area. The program's single menu item allows access to options which toggle various attributes of the movie controller. For example, selecting **Hide Controller** makes the entire movie controller invisible. Clicking **Hide Step Buttons**, **Hide Speaker Button** or **Hide Grow Box** removes these elements from the controller. The other options are equally self-explanatory, and it is a good idea to play around with them to see how they work.

# E. FILTERS - Using Action Filters

# 1. Introduction

Action filters are the means by which you can customize movie controller behavior. When you set a filter, all subsequent MCDoAction calls will immediately call your filter function, giving you first crack at handling the action specified by MCDoAction. In Windows terms, you are essentially subclassing a movie controller. Additionally, your filter can tell MCDoAction to return immediately or pass the action through to the controller for normal processing.

FILTERS.EXE intercepts incoming movie controller bounds rectangle change messages (resulting, for example, from dragging the grow box) and then resizes the movie rectangle proportionately, i.e. preserving the original aspect ratio. The resulting bounds rectangle is scaled proportionately, adjusting the height to match the width to which it has been dragged.

## 2. The FILTERS Source Code

```
FILTERS.MAK

ALL : FILTERS.EXE

FILTERS.OBJ : FILTERS.C
 cl -c -AS -DSTRICT -G2 -GA -GEs -Zpel -W3 -WX -Od filters.c

FILTERS.EXE : FILTERS.OBJ FILTERS.DEF
 link /nod /a:16 filters, filters.exe, nul, qtw libw slibcew, \
    filters.def;
 rc filters.exe


FILTERS.DEF

NAME        FILTERS
DESCRIPTION 'Sample Application'
EXETYPE     WINDOWS
STUB        'winstub.exe'
CODE        PRELOAD MOVEABLE DISCARDABLE
DATA        PRELOAD MOVEABLE MULTIPLE
HEAPSIZE    1024
STACKSIZE   16384


FILTERS.C

#include <windows.h>
#include <qtw.h>

long FAR PASCAL __export WndProc  (HWND, UINT, WPARAM, LPARAM);
BOOL CALLBACK __export TestFilter (MovieController, UINT,
   LPVOID, LONG);

MovieController mcController;
RECT rcNorm;
SHORT sMCHeight;

int PASCAL WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
   LPSTR lpszCmdParam, int nCmdShow)
   {
   static char szAppName[] = "Filters";
   HWND        hWnd;
   MSG         msg;
   WNDCLASS    wndclass;
   MovieFile   mfMovie;
   RECT        rcMovie;
   Movie       mMovie;

// Establish links to QuickTime for Windows
   if (QTInitialize (NULL))
      {
      MessageBox (NULL, "QTInitialize failure", szAppName, MB_OK);
```

```
      return 0;
      }
// Allocate memory required for playing movies
   if (EnterMovies ())
      {
      MessageBox (NULL, "EnterMovies failure", szAppName, MB_OK);
      return 0;
      }
// Register and create main window
   if (!hPrevInstance)
      {
      wndclass.style         = CS_DBLCLKS | CS_HREDRAW | CS_VREDRAW;
      wndclass.lpfnWndProc   = WndProc;
      wndclass.cbClsExtra    = 0;
      wndclass.cbWndExtra    = 0;
      wndclass.hInstance     = hInstance;
      wndclass.hIcon         = LoadIcon (NULL,IDI_APPLICATION);
      wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW);
      wndclass.hbrBackground = (HBRUSH) (COLOR_WINDOW + 1);
      wndclass.lpszMenuName  = NULL;
      wndclass.lpszClassName = szAppName;
      if (!RegisterClass (&wndclass))
         {
         MessageBox (NULL, "RegisterClass failure", szAppName, MB_OK);
         return 0;
         }
      }
   hWnd = CreateWindow(szAppName, szAppName, WS_OVERLAPPEDWINDOW |
      WS_CLIPCHILDREN, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,
      CW_USEDEFAULT, NULL, NULL, hInstance, NULL);
   if (hWnd == NULL)
      {
      MessageBox (NULL, "CreateWindow failure", szAppName, MB_OK);
      return 0;
      }
// Instantiate the movie
   if (OpenMovieFile ("SAMPLE.MOV", &mfMovie, OF_READ) != noErr)
      {
      MessageBox (NULL, "OpenMovieFile failure", szAppName, MB_OK);
      return 0;
      }
   NewMovieFromFile (&mMovie, mfMovie, NULL, NULL, 0, NULL);
   CloseMovieFile (mfMovie);
// Get the normal movie dimensions. We'll use these as the
// movie aspect ratio in the filter
   GetMovieBox (mMovie, &rcNorm);
   OffsetRect (&rcNorm, -rcNorm.left, -rcNorm.top);
// Build the movie rectangle
   GetClientRect (hWnd, &rcMovie);
   rcMovie.top = (rcMovie.bottom / 3) - (rcNorm.bottom / 2);
   rcMovie.bottom = rcMovie.top + rcNorm.bottom;
   rcMovie.left = (rcMovie.right / 3) - (rcNorm.right / 2);
   rcMovie.right = rcMovie.left + rcNorm.right;
// Instantiate the movie controller
   mcController = NewMovieController (mMovie, &rcMovie,
      mcTopLeftMovie + mcScaleMovieToFit + mcWithBadge, hWnd);
// Make the movie paused initially
```

```
   MCDoAction (mcController, mcActionPlay, 0);
// Calculate the controller height for use in filter
   MCGetControllerBoundsRect (mcController, &rcMovie);
   OffsetRect (&rcMovie, -rcMovie.left, -rcMovie.top);
   sMCHeight = rcMovie.bottom - rcNorm.bottom;
// Set an action filter, passing in the parent window handle
   MCSetActionFilterMCSetActionFilter (mcController, TestFilter, (LONG)
((LPVOID) hWnd));
// Enable the keyboard interface
   MCDoAction (mcController, mcActionSetKeysEnabled, (LPVOID) TRUE);
// Make the movie active
   SetMovieActive (mMovie, TRUE);
// Make the main window visible
   ShowWindow (hWnd, nCmdShow);
   UpdateWindow (hWnd);
// Play the movie
   while (GetMessage (&msg, NULL, 0, 0))
      {
      TranslateMessage (&msg);
      DispatchMessage (&msg);
      }
// Destroy the movie controller
   DisposeMovieController (mcController);
// Destroy the movie
   DisposeMovie (mMovie);
// Cut the connections to QuickTime for Windows
   ExitMovies ();
   QTTerminate ();
// Return to Windows
   return msg.wParam;
   }
long FAR PASCAL WndProc (HWND hWnd, UINT message, WPARAM wParam,
   LPARAM lParam)
   {
   PAINTSTRUCT ps;
// Drive the movie controller
   if (MCIsPlayerMessage (mcController, hWnd, message, wParam, lParam))
      return 0;
// Process the windows message
   switch (message)
      {
      case WM_PAINT:
         if (!BeginPaint (hWnd, &ps))
            return 0;
         EndPaint (hWnd, &ps);
         return 0;
      case WM_DESTROY:
         PostQuitMessage (0);
         return 0;
      }
// Return to Windows
   return DefWindowProc (hWnd, message, wParam, lParam);
   }

BOOL CALLBACK __export TestFilter (MovieController mcCaller,
   UINT uAction, LPVOID lpParam, LONG refcon)
   {
```

```
   RECT rcBounds;
   static BOOL bBlock;

// Don't want to recursively call ourselves
   if (bBlock)
      return FALSE;
// Respond to mcAction
   switch (uAction)
      {
      case mcActionControllerSizeChanged:
      // Force a paint of the old client rectangle
         InvalidateRect ((HWND) refcon, NULL, TRUE);
         MCGetControllerBoundsRect (mcCaller, &rcBounds);
      // Calculate new bounds rect bottom
         rcBounds.bottom =
            rcBounds.top + MulDiv (rcBounds.right - rcBounds.left,
               rcNorm.bottom, rcNorm.right);
      // Add the controller height back in
         rcBounds.bottom += sMCHeight;
         bBlock = TRUE;
         MCSetControllerBoundsRect (mcCaller, &rcBounds);
         bBlock = FALSE;
         return TRUE;
      default:
         return FALSE;
      }
   }
```

## 3. Declaring an Action Filter

Each movie controller in your program can have a unique action filter, but only one at a time. To be used successfully, an action filter must meet certain criteria:

- It must be a callback function
- It must be explicitly exported
- It must use a defined parameter list.

FILTERS uses an action filter named `TestFilter`:

```
BOOL CALLBACK __export TestFilter (MovieController, UINT FAR *,
   LPVOID, LONG);
```

Like normal window or dialog procedures, it is declared as `CALLBACK`. It returns a boolean value denoting whether the action passed to it by MCDoAction should be processed normally when the filter returns (`FALSE`), or if `MCDoAction` should itself return at that point (`TRUE`).

The filter's first argument is the related movie controller object. Its second is the address of the *mcAction* item currently being handled by MCDoAction. The third is an additional value dependent on the second. Fourth is a variable for passing additional data to the filter. The first three arguments are essentially a pass-through of the parameters passed to `MCDoAction` when it was called.

## 4. Setting an Action Filter

The routine used to set an action filter is MCSetActionFilter:

```
HANDLE hInst;
MovieController mcController;
•
•
MCSetActionFilter (mcController, TestFilter, 0L);
```

You can set a new action filter at any time in your program. If you want to remove a filter, you must call MCSetActionFilter with a `NULL` filter parameter:

```
HANDLE hInst;
MovieController mcController;
•
•
MCSetActionFilter (mcController, (MCActionFilter) NULL, 0L);
```

Although not demonstrated above, the last parameter can be used to pass data such as a window handle or the address of a structure with useful information for the action filter. Filter functions may be defined in any of your application's modules, either the executable itself or a library.

## 5. Defining an Action Filter

The action filter used by FILTERS traps dragging the grow box. If you wished, you could code cases for all of the possible *mcActions* and create unusual behavior for each. The filter would still function normally, although your movie might not perform as well as expected. In other words, if your program needs a filter, be sure to plan carefully for all of the extra processing that will be involved.

The basic layout of a filter is similar to a window procedure. One difference to note is that the action parameter is actually the address of the action item.

```
BOOL CALLBACK __export TestFilter (MovieController mcCaller,
   UINT uAction, LPVOID lpParam, LONG lRefCon)
   {
   switch (uAction)
      {
      /* cases */
      }
   return FALSE;
   }
```

Each of your cases should return `TRUE` or `FALSE` when its processing is finished. Good QuickTime for Windows style specifies that the default return value be `FALSE`, causing the action to be handled normally by the movie controller if the filter didn't process anything. You should also note that you can dynamically change the action your filter is switching on, since you have received its address. This flexibility can be advantageous when you want to fall through to normal processing with a new *mcAction*:

```
switch (uAction)
      {
      case mcActionControllerSizeChanged:

         uAction = mcActionBadgeClick;
         return FALSE;
      •
      •
      }
```

The case `TestFilter` deals with is resizing the bounds rectangle if the grow box is dragged. This causes QuickTime for Windows to generate a MCDoAction all with an *mcAction* of mcActionControllerSizeChanged. The third parameter, `lParam`, has no bearing on this particular action and is not handled. `TestFilter`'s last argument, `lRefCon`, receives the application's parent window handle so the filter can call `InvalidateRect`.

```
case mcActionControllerSizeChanged:
// Force a paint of the old client rectangle
   InvalidateRect ((HWND) refcon, NULL, TRUE);
   MCGetControllerBoundsRect (mcCaller, &rcBounds);
// Calculate new bounds rect bottom
   rcBounds.bottom =
      rcBounds.top + MulDiv (rcBounds.right - rcBounds.left,
         rcNorm.bottom, rcNorm.right);
// Add the controller height back in
   rcBounds.bottom += sMCHeight;
   bBlock = TRUE;
   MCSetControllerBoundsRect (mcCaller, &rcBounds);
   bBlock = FALSE;
   return TRUE;
```

When our grow box is dragged and released, QuickTime for Windows recalculates the controller's bounds

rectangle. In this simplified example, we first ensure that no garbage is left on the screen by calling `InvalidateRect`. We then retrieve the new rectangle with MCGetControllerBoundsRect. After subtracting the height of the movie controller derived in `WinMain`, we calculate a new depth for our movie based on its new width. The effect is to vary the height to preserve the original aspect ratio of the movie. Calling MCSetControllerBoundsRect displays the adjusted rectangle.

In general, if your application contains a movie controller with a grow box, you should use a filter to let the program know when the controller's size or position changes, since the program has no other way of knowing when this happens (you may have observed the consequences in BIGEIGHT). By providing such a filter, you can allow, say, a word processor to flow its text around a redimensioned movie, or simply let a program such as FILTERS clean up after itself.

# A. QuickTime for Windows API - Functions

MCDoAction mcActionDeactivate
MCDoAction mcActionDraw
MCDoAction mcActionGetFlags
MCDoAction mcActionGetKeysEnabled
MCDoAction mcActionGetLooping
MCDoAction mcActionGetLoopIsPalindrome
MCDoAction mcActionGetPlayEveryFrame
MCDoAction mcActionGetPlayRate
MCDoAction mcActionGetPlaySelection
MCDoAction mcActionGetUseBadge
MCDoAction mcActionGetVolume
MCDoAction mcActionGoToTime
MCDoAction mcActionIdle
MCDoAction mcActionKey
MCDoAction mcActionPlay
MCDoAction mcActionSetFlags
MCDoAction mcActionSetGrowBoxBounds
MCDoAction mcActionSetKeysEnabled
MCDoAction mcActionSetLooping
MCDoAction mcActionSetLoopIsPalindrome
MCDoAction mcActionSetPlayEveryFrame
MCDoAction mcActionSetPlaySelection
MCDoAction mcActionSetSelectionBegin
MCDoAction mcActionSetSelectionDuration
MCDoAction mcActionSetUseBadge
MCDoAction mcActionSetVolume
MCDoAction mcActionStep
MCDraw
MCDrawBadge
MCGetControllerBoundsRect
MCGetControllerInfo
MCGetCurrentTime
MCGetMovie
MCGetVisible
MCIdle
MCIsControllerAttached
MCIsPlayerMessage
MCKey
MCNewAttachedController
MCPositionController
MCSetActionFilter
MCSetControllerAttached
MCSetControllerBoundsRect
MCSetMovie
MCSetVisible
NewMovieController
NewMovieFromDataFork
NewMovieFromFile
NormalizeRect
OpenMovieFile
OpenPictureFile
PictureToDIB
PrerollMovie
PtInMovie
QTFOURCC
QTInitialize

QTTerminate
SetMovieActive
SetMovieCoverProcs
SubtractTime
UpdateMovie

## *AddTime*

Syntax

```
VOID AddTime (TimeRecord FAR *lptrDst, const TimeRecord  FAR *lptrSrc)
```

`AddTime` adds two time records together, replacing the first with the result. A TimeRecord is a structure that references a particular point in a movie, or a duration within a movie.

Parameters

TimeRecord *FAR* `*lptrDst`

The address of a time record containing the first operand for the addition. The TimeRecord referenced is overwritten by the result of the addition.

*const* TimeRecord *FAR* `*lptrSrc`

The address of a time record containing the second operand for the addition. The TimeRecord referenced remains unmodified by the operation.

Return

None. The result is placed in the time record referenced by the first parameter. Use GetMoviesError and GetMoviesStickyError to test for failure of this call.

Comments

If the time records contain different time scales, `AddTime` converts them as appropriate.

Example

```
MovieController mcController;
TimeRecord trOne, trTwo;
•
•
AddTime (&trOne, &trTwo);
MCDoAction (mcController, mcActionGoToTime, (LPVOID) &trOne);
```

See Also:

Functions

ConvertTimeScale, GetMovieTimeScale, SubtractTime, GetMoviesError, GetMoviesStickyError

MCDoAction

mcActionGoToTime

Data Types

TimeRecord, `TimeValue`

## *ClearMoviesStickyError*

Syntax

```
VOID ClearMoviesStickyError (VOID)
```

`ClearMoviesStickyError` clears the sticky error value. The sticky error value is the first non-zero error code returned by an eligible QuickTime for Windows routine since `ClearMoviesStickyError` was last called. Eligible QuickTime for Windows routines operate on movies (as opposed to movie controllers) and require a movie object.

Parameters This routine takes no parameters.

Return None.

Comments A result code is not placed into the sticky error value until the field has been cleared. Your application should clear the sticky error value when necessary to ensure that it does not contain a stale result code.

Example

```
Movie mMovie;
LFIXED lfxRate;
   •
   •
// Assume call produces an error code

   lfxRate = GetMoviePreferredRate (mMovie);

// Assume other calls follow with no errors
   •
   •
   if (GetMoviesStickyError())
      {
      MessageBox (NULL, "GetMoviePreferredRate Failure",
         "Program", MB_OK);
      ClearMoviesStickyError ();
      }
```

See Also:

Functions GetMoviesError, GetMoviesStickyError

## *CloseMovieFile*

Syntax `OSErr CloseMovieFile (MovieFile mfMovie)`

`CloseMovieFile` closes an open movie file.

Parameters *`MovieFile`* `mfMovie`

The reference value assigned by OpenMovieFile.

Return Zero if no error condition. Non-zero if error condition. See Appendix A for error condition values. You can also use GetMoviesError and GetMoviesStickyError to test for failure of this call.

Comments It is good QuickTime for Windows programming style to close an opened movie file at the first opportunity, e.g. once the movie object has been extracted.

Example

```
MovieFile mfMovie;
Movie mMovie;
•
```

```
•
OpenMovieFile ("SAMPLE.MOV", &mfMovie, OF_READ);
NewMovieFromFile (&mMovie, mfMovie, NULL, NULL, 0, NULL);
CloseMovieFile (mfMovie);
```

See Also:

Functions OpenMovieFile, GetMoviesError, GetMoviesStickyError

### *ClosePictureFile*

Syntax `OSErr ClosePictureFile (PicFile pfPicture)`

`ClosePictureFile` closes an open picture file.

Parameters *`PicFile`* `pfPicture`

The reference value assigned by OpenPictureFile.

Return Zero if no error condition. Non-zero if error condition. See Appendix A for error condition values. You can also use GetMoviesError and GetMoviesStickyError to test for failure of this call.

Comments It is good QuickTime for Windows programming style to close an opened picture file at the first opportunity, e.g. once the necessary data has been extracted.

Example

```
PicFile pfPicture;
•
•
if (OpenPictureFile ("PICTURE.PIC", &pfPicture, OF_READ))
   {
   /* Inform user of failure. */
   }
•
•
ClosePictureFile (pfPicture);
```

See Also:

Functions OpenPictureFile, GetMoviesError, GetMoviesStickyError

### *ConvertTimeScale*

Syntax `VOID ConvertTimeScale (TimeRecord FAR *lptrInout, TimeScale tsNewScale)`

`ConvertTimeScale` converts a time from one time scale into a time relative to another time scale.

Parameters TimeRecord *`FAR`* `*lptrInout`

A pointer to a TimeRecord which you must populate with the `TimeValue` and the `TimeScale` you wish to convert.

*`TimeScale`* `tsNewScale`

The `TimeScale` to which you wish to convert.

Return

None. The TimeRecord referenced by the first parameter is overwritten with the converted `TimeValue` and `TimeScale` values that were the basis of the conversion. Use GetMoviesError and GetMoviesStickyError to test for failure of this call.

Comments

The time coordinate system contains a time scale scored in time units. The number of units that pass per second quantifies the scale: a time scale of 26 means that 26 units pass per second and each time unit is 1/26 of a second.

When the duration of all or part of a movie is needed, it is expressed as a number of time units. Particular points in a movie can be identified by a time value, which is the number of time units to that point from the beginning of the movie.

Different movies may have different time scales. Use `ConvertTimeScale` to compare TimeValues between different movies.

Example

```
Movie mMovieA, MovieB;
TimeRecord trRecord;
•
•
// Convert a TimeValue in Movie A to its TimeValue in Movie B

   trRecord.value.dwLo = GetMoviePosterTime (mMovieA);
   trRecord.value.dwHi = 0;
   trRecord.scale = GetMovieTimeScale (mMovieA);
   ConvertTimeScale (&trRecord, GetMovieTimeScale (mMovieB));
```

See Also:

Functions

GetMovieDuration, GetMovieTimeScale, MCGetCurrentTime, GetMoviesError, GetMoviesStickyError

Data Types

TimeRecord, `TimeValue`

## *CountUserDataType*

Syntax

```
LONG CountUserDataType (UserData udData, OSType ostType)
```

`CountUserDataType` determines the number of items of a given type in a user data list.

Parameters

*`UserData`* `udData`

The handle to the user data list.

*`OSType`* `ostType`

The user data type.

Return — The number of items of the specified type in the user data list. You can use GetMoviesError and GetMoviesStickyError to test for failure of this call.

Comments — A movie's user data list is placed in a movie by its creator and may contain items of various types. A common type is text containing copyright data, names of people involved in the movie's production, special hardware and software requirements, and other types of information about the movie. By convention, text user data types start with a "©" symbol.

Example — See the example in the description of GetUserDataText.

See Also:

Functions — GetMovieUserData, GetNextUserDataType, GetUserData, GetUserDataText, GetMoviesError, GetMoviesStickyError

Data Types — `UserData, OSType`

## *CoverProc*

Syntax

```
OSErr CALLBACK CoverProc (Movie mMovie, HDC hdc, LONG lID)
```

`CoverProc` is the prototype for the cover (or uncover) procedure set by the routine SetMovieCoverProcs. It shows the parameters you must pass to your cover procedure, and the value the procedure must return.

Parameters

*Movie* `mMovie`

The movie object.

*HDC* `hdc`

The handle to a device context, whose clipping region is preset to the area being covered or uncovered.

*LONG* `lID`

The reference constant supplied in the SetMovieCoverProcs call. You can use this value to allow a single cover procedure to handle multiple cases.

Return — Your cover procedure should return `noErr` if it does not detect an error. Otherwise, return one of the values defined in Appendix A.

Comments — `CoverProc` is not a defined QuickTime for Windows function. It is a prototype only, used as a template for your cover procedures.

Example

```
OSErr CALLBACK __export MyCoverProc (Movie, HDC, LONG);
```

```
•
HWND hWnd;
Movie mMovie;
•
•
SetMovieCoverProcs (mMovie, MyCoverProc, NULL, 5879);
•
•
OSErr CALLBACK __export MyCoverProc (Movie m, HDC hdc, lID)
   {
   RECT rcClip;
   GetClipBox (hdc, &rcClip);
   FillRect (hdc, &rcClip, GetStockObject (WHITE_BRUSH));
   return 0;
   }
```

See Also:

Functions SetMovieCoverProcs

## *DeleteMovieFile*

Syntax `OSErr DeleteMovieFile (LPCSTR lpstrFileSpec)`

`DeleteMovieFile` deletes a movie file.

Parameters *`LPCSTR`* `lpstrFileSpec`

The name of the movie file, including the extension (.MOV).

Return Zero if no error condition. Non-zero if error condition. See Appendix A for error condition values. You can also use GetMoviesError and GetMoviesStickyError to test for failure of this call.

Comments Physically deletes a movie file from the disk media.

Example
```
DeleteMovieFile ("NEWSREEL.MOV");
```

See Also:

Functions OpenMovieFile, CloseMovieFile, GetMoviesError, GetMoviesStickyError

## *DisposeMovie*

Syntax `VOID DisposeMovie (Movie mMovie)`

`DisposeMovie` frees any memory being used by a movie. Your program should call this routine when it is done working with a movie.

Parameters *`Movie`* `mMovie`

The movie object whose memory is being released.

Return None. Use GetMoviesError and GetMoviesStickyError to test for failure of this call.

Comments `DisposeMovie` must be called, ultimately, for each movie instantiated by your program. It does not affect the DOS file containing the movie or the movie controller to which it may be attached.

Example
```
Movie mMovie;
MovieFile mfMovie;
•
•
OpenMovieFile ("SAMPLE.MOV", &mfMovie, OF_READ);
NewMovieFromFile (&mMovie, mfMovie, NULL, NULL, 0, NULL);
CloseMovieFile (mfMovie);
•
•
DisposeMovie (mMovie);
```

See Also:

Functions NewMovieFromFile, DisposeMovieController, GetMoviesError, GetMoviesStickyError

## *DisposeMovieController*

Syntax `VOID DisposeMovieController (MovieController mcController)`

`DisposeMovieController` destroys a movie controller.

Parameters *`MovieController`* `mcController`

The movie controller object being destroyed.

Return None.

Comments `DisposeMovieController` must be called, ultimately, for every movie controller created by your program. This function does not affect any movie associated with the controller being destroyed.

Example
```
MovieController mcController;
Movie mMovie;
RECT rcMovie;
HWND hWnd;
•
•
mcController = NewMovieController (mMovie, &rcMovie,
   mcTopLeftMovie, hWnd);
•
•
DisposeMovieController (mcController);
```

See Also:

Functions NewMovieController, DisposeMovie

## *DisposePicture*

Syntax `VOID DisposePicture (PicHandle phPicture)`

`DisposePicture` frees any memory being used by a QuickTime for Windows picture. Your program should call this routine when it is done working with a QuickTime for Windows picture.

Parameters *`PicHandle`* `phPicture`

The picture object whose memory is being released.

Return None. Use GetMoviesError and GetMoviesStickyError to test for failure of this call.

Comments Either KillPicture or `DisposePicture` must be called, ultimately, for each picture instantiated by your program. It does not affect the DOS file containing the picture.

Example

```
PicHandle phPicture;
PicFile pfPicture;
•
•
if (!OpenPictureFile ("PICTURE.PIC", &pfPicture, OF_READ))
   {
   phPicture = GetPictureFromFile (pfPicture);
   ClosePictureFile (pfPicture);
   }
•
•
DisposePicture (phPicture);
```

See Also:

Functions GetPictureFromFile, OpenPictureFile, ClosePictureFile, KillPicture, GetMoviesError, GetMoviesStickyError

## *DrawPicture*

Syntax `OSErr DrawPicture (HDC hdc, PicHandle phThePict, const LPRECT lprcFrame, ProgressProcRecordPtr pprpProgressProc))`

`DrawPicture` draws a picture in the QuickTime for Windows format.

Parameters *`HDC`* `hdc`

The handle to the device context.

*`PicHandle`* `phThePict`

The picture object.

*const LPRECT* lprcFrame

The address of a rectangle in which the picture is to be drawn (in client area coordinates).

*ProgressProcRecordPtr* pprpProgressProc

Reserved. Should be coded as NULL.

Return

Zero if no error condition. Non-zero if error condition. See Appendix A for error condition values. You can also use GetMoviesError and GetMoviesStickyError to test for failure of this call.

Comments

A picture is a still image held in memory (e.g. a frame from a movie), in a format usable by QuickTime for Windows. A PicHandle is an object reference to this type of image, obtained by a call such as GetMoviePict (see the description of this routine). The picture object must be freed when you are done with it. *Note: All QuickTime for Windows routines referencing a RECT or POINT assume client device coordinates.*

Example

```
Movie mMovie;
MovieController mcController;
PicHandle phPicture;
RECT rcPicture;
HDC hdc;
TimeValue tvTime;
•
•
// Retrieve last movie frame, display it at different location

   tvTime = GetMovieDuration (mMovie);
   if ((phPicture = GetMoviePict (mMovie, tvTime)) != NULL)
      DrawPicture (hdc, phPicture, &rcPicture, NULL);
•
•
// Don't forget to free the picture object

   DisposePicture (phPicture);
```

See Also:

Functions

GetMoviePict, PictureToDIB, GetMoviesError, GetMoviesStickyError

DataTypes

PicHandle

## *DrawPictureFile*

Syntax

```
OSErr DrawPictureFile (HDC hdc, PicFile pfPicture, const LPRECT
lprcFrame, ProgressProcRecordPtr pprpProgressProc)
```

DrawPictureFile draws an image from the specified picture file.

Parameters

*HDC* hdc

A handle to the device context.

*PicFile* pfPicture

The picture file reference value returned by OpenPictureFile.

*const LPRECT* lprcFrame

A pointer to a rectangle where the picture is to be drawn (in client area coordinates).

*ProgressProcRecordPtr* pprpProgressProc

Reserved. Should be coded as NULL.

Return

Zero if no error condition. Non-zero if error condition. See Appendix A for error condition values. You can also use GetMoviesError and GetMoviesStickyError to test for failure of this call.

Comments

This function is essentially the same as the DrawPicture function, except that it reads the picture from disk. Picture files are characterized by the DOS file suffix ".PIC", and are DOS versions of Macintosh PICT and JFIF files. *Note: All QuickTime for Windows routines referencing a RECT or POINT assume client device coordinates.*

Example

```
PicFile pfPicture;
RECT rcPict;
HDC hdc;
•
•
OpenPictureFile ("HOUSE.PIC", &pfPicture, OF_READ);
DrawPictureFile (hdc, pfPicture, &rcPict, NULL);
ClosePictureFile (pfPicture);
```

See Also:

Functions

ClosePictureFile, DrawPicture, GetPictureFileInfo, GetPictureInfo, GetMoviesError, GetMoviesStickyError, OpenPictureFile

## *EnterMovies*

Syntax

```
OSErr EnterMovies (VOID)
```

`EnterMovies` allocates memory for QuickTime for Windows to run itself.

Parameters

This function takes no parameters.

Return

Zero if no error condition. Non-zero if error condition. See Appendix A for error condition values.

Comments

`EnterMovies` only needs to be called once during the life of your programs that play

movies. The memory allocated is not memory used for movies, but rather for global QuickTime for Windows activities. An application may call `EnterMovies` multiple times, but storage will only be allocated the first time.

Example

```
if (EnterMovies() != noErr)
   {
   MessageBox (NULL, "EnterMovies failure", "WinPlay1",
      MB_OK);
   return 0;
   }
```

See Also:

Functions ExitMovies, QTInitialize, QTTerminate

## *ExitMovies*

Syntax `VOID ExitMovies (VOID)`

`ExitMovies` frees memory used by QuickTime for Windows to run itself.

Parameters This routine takes no parameters.

Return None.

Comments The memory released is the global memory used by QuickTime for Windows. It is not the memory used to store movies. QuickTime for Windows programs that do not call EnterMovies (e.g. those that display only individual QuickTime for Windows pictures) do not have to call `ExitMovies`.

Example

```
// Cut the connections to QuickTime for Windows

   ExitMovies ();
   QTTerminate ();
```

See Also:

Functions EnterMovies, QTInitialize, QTTerminate

## *GetMovieActive*

Syntax `BOOL GetMovieActive (Movie mMovie)`

`GetMovieActive` queries the active state of a movie (whether or not it can be played).

Parameters *`Movie`* `mMovie`

The movie object.

Return `TRUE` if the movie is active. `FALSE` if the movie is inactive. You can use GetMoviesError and GetMoviesStickyError to test for failure of this call.

Comments A movie with an inactive state will remain visible but will not play, since it does not receive cycles from QuickTime for Windows' scheduler while inactive.

Do not confuse a movie's active state with its playing/paused state, i.e. do not use SetMovieActive to start or stop playing a movie. You can set a movie's active state using SetMovieActive.

Example

```
Movie mMovie;
•
•
// If the movie is active, make it inactive

   if (GetMovieActive (mMovie))
      {
      SetMovieActive (mMovie, FALSE);
      }
```

See Also:

Functions SetMovieActive, GetMoviesError, GetMoviesStickyError

## *GetMovieActiveSegment*

Syntax

```
VOID GetMovieActiveSegment (Movie mMovie, TimeValue FAR *,
TimeValue FAR *)
```

`GetMovieActiveSegment` determines which segment of a movie is currently selected for playing.

Parameters

*`Movie`* `mMovie`

The movie object.

*`TimeValue FAR`* `*tvStart`

A pointer to the start time value.

*`TimeValue FAR`* `*tvDuration`

A pointer to the duration time value.

Return `tvStart` and `tvDuration` are populated with the starting time and the duration of the active movie segment, respectively. You can use GetMoviesError and GetMoviesStickyError to test for failure of this call.

Comments If the active segment is the entire movie, `tvStart` is set to -1 and `tvDuration` is set to zero.

Example

```
Movie mMovie;
```

```
TimeValue tvStart, tvDuration;
•
•
GetMovieActiveSegment (mMovie, &tvStart, &tvDuration);
if (tvStart == -1)
    /* Code for when entire movie is active. */
else
    /* Code for when subset of entire movie is active. */
```

See Also:

Functions GetMovieActive, MCDoAction, GetMoviesError, GetMoviesStickyError

MCDoAction mcActionSetSelectionBegin, mcActionSetPlaySelection, mcActionSetSelectionDuration

## *GetMovieBox*

Syntax `VOID GetMovieBox (Movie mMovie, LPRECT lprcMovieRect)`

`GetMovieBox` obtains the current dimensions of a movie rectangle.

Parameters *`Movie`* `mMovie`

The movie object.

*`LPRECT`* `lprcMovieRect`

The address of the movie rectangle.

Return The rectangle referenced by `lprcMovieRect` is populated with the movie's current dimensions. Use GetMoviesError and GetMoviesStickyError to test for failure of this call.

Comments The movie need not be visible on the screen for this function to provide its dimensions. Consequently, this call is quite useful for determining the optimum rectangle for displaying a movie when calling NewMovieController.

If the rectangle referenced by `lprcMovieRect` is NULL, a sound-only movie is indicated. It is up to you to handle this condition however you wish.

*Note: All QuickTime for Windows routines referencing a RECT or POINT assume client device coordinates.*

Example

```
RECT rcMovie;
Movie mMovie;
MovieFile mfMovie;
MovieController mcController;
•
•
// Open the movie file

   if (OpenMovieFile ("NEWSREEL.MOV", &mfMovie, OF_READ))
      {
      MessageBox (NULL, "Open failure", ...);
```

```
        }
    NewMovieFromFile (&mMovie, mfMovie, NULL, NULL,
        newMovieActive, NULL);
    CloseMovieFile  (mfMovie);

// Instantiate the movie controller

    GetMovieBox (mMovie, &rcMovie);
    OffsetRect (&rcMovie, -rcMovie.left, -rcMovie.top);
    mcController = NewMovieController (mMovie, &rcMovie,
        mcTopLeftMovie + mcScaleMovieToFit, hWnd);
```

See Also:

Functions MCGetControllerBoundsRect, GetMoviesError, GetMoviesStickyError

## *GetMovieCreationTime*

Syntax `LONG GetMovieCreationTime (Movie mMovie)`

`GetMovieCreationTime` retrieves a movie's creation date and time.

Parameters *`Movie`* `mMovie`

The movie object.

Return A `LONG` containing the movie's creation date and time information. You can use GetMoviesError and GetMoviesStickyError to test for failure of this call.

Comments The returned `LONG` may be decoded using the C language `ctime` function.

Example

```
LONG lDateTime;
Movie mMovie;
char buffer [80];
•
•
lDateTime = GetMovieCreationTime (mMovie);
wsprintf (buffer, "Movie created on %s", ctime (&lDateTime));
```

See Also:

Functions GetMovieModificationTime, GetMoviesError, GetMoviesStickyError

## *GetMovieDataSize*

Syntax `LONG GetMovieDataSize (Movie mMovie, TimeValue tvStart, TimeValue tvDuration)`

`GetMovieDataSize` retrieves the size, in bytes, of the data in a segment of a movie. This size includes both video and sound data.

Parameters *Movie* mMovie

The movie object.

*TimeValue* tvStart

A time value specifying the starting point of the segment whose size is being queried.

*TimeValue* tvDuration

A time value specifying the duration of the segment whose size is being queried.

Return A LONG that contains the size, in bytes, of the movie's data that lies in the specified segment. Use GetMoviesError and GetMoviesStickyError to test for failure of this call.

Comments This function may be called whether a movie is playing or not. Use MCGetCurrentTime to retrieve the movie's current time.

Example

```
LONG lSize;
Movie mMovie;
TimeValue tvStart, tvDuration;
MovieController mcController;
•
•
// Get the number of bytes from the current position to two
// seconds later

   tvStart = MCGetCurrentTime (mcController, NULL);
   tvDuration = 2 * GetMovieTimeScale (mMovie);
   lSize = GetMovieDataSize (mMovie, tvStart, tvDuration);
```

See Also:

Functions ConvertTimeScale, MCGetCurrentTime, GetMoviesError, GetMoviesStickyError, GetMovieTimeScale

Data Types TimeValue

## *GetMovieDuration*

Syntax `TimeValue GetMovieDuration (Movie mMovie)`

GetMovieDuration retrieves the duration of a movie, expressed in units of the movie's time scale.

Parameters *Movie* mMovie

The movie object.

Return A TimeValue containing the movie's duration, in units of the movie's time scale. Use GetMoviesError and GetMoviesStickyError to test for failure of this call.

Comments

QuickTime for Windows' time coordinate system uses a time scale scored in time units. The number of units that pass per second quantifies the scale: a time scale of 26 means that 26 units pass per second and each time unit is 1/26 of a second.

When the duration of all or part of a movie is needed, it is expressed in the number of time units it contains. Particular points in a movie can be identified by time values, which are the number of time units to those points from the beginning of the movie.

Different movies may have different time scales. Use ConvertTimeScale to compare `TimeValues` between differently scaled movies.

Example

```
LONG lSize;
Movie mMovie;
TimeValue tvDuration;
•
•
// Get the number of bytes in this movie

   tvDuration = GetMovieDuration (mMovie);
   lSize = GetMovieDataSize (mMovie, 0, tvDuration);
```

See Also:

Functions ConvertTimeScale, GetMovieTimeScale, MCGetCurrentTime, GetMoviesError, GetMoviesStickyError

Data Types `TimeValue`

### *GetMovieModificationTime*

Syntax `LONG GetMovieModificationTime (Movie mMovie)`

`GetMovieModificationTime` retrieves a movie's last modification date and time.

Parameters *`Movie`* `mMovie`

The movie object.

Return A `LONG` containing the movie's last modification date and time. You can use GetMoviesError and GetMoviesStickyError to test for failure of this call.

Comments The resulting `LONG` may be decoded using the C language `ctime` function.

Example

```
LONG lDateTime;
Movie mMovie;
char buffer [80];
•
•
lDateTime = GetMovieModificationTime (mMovie);
sprintf (buffer, "Movie modified on %s", ctime (&lDateTime));
```

See Also:

Functions GetMovieCreationTime, GetMoviesError, GetMoviesStickyError

## *GetMoviePict*

Syntax `PicHandle GetMoviePict (Movie mMovie, TimeValue tvTime)`

`GetMoviePict` retrieves an individual image from a movie in the QuickTime for Windows picture format at a specified movie time.

Parameters *`Movie`* `mMovie`

The movie object.

*`TimeValue`* `tvTime`

The time value in the movie of the image to be retrieved.

Return A picture object. A NULL return indicates failure. You can also use GetMoviesError and GetMoviesStickyError to test for failure of this call.

Comments This function may be called whether a movie is playing or not. The picture object returned is unusable by Windows directly. Use the function PictureToDIB to convert the image to a Windows Device Independent Bitmap (DIB). An alternative to converting the image is using DrawPicture to display it at specified coordinates.

Example

```
Movie mMovie;
MovieController mcController;
PicHandle phPicture;
RECT rcPicture;
HDC hdc;
TimeValue tvTime;
•
•
// Retrieve last movie frame then display it on the
// screen at another location

   tvTime = GetMovieDuration (mMovie);
   if ((phPicture = GetMoviePict (mMovie, tvTime)) != NULL)
      DrawPicture (hdc, phPicture, &rcPicture, NULL);
```

See Also:

Functions DrawPicture, GetMoviePosterPict, MCGetCurrentTime, PictureToDIB, GetMoviesError, GetMoviesStickyError

## *GetMoviePosterPict*

Syntax `PicHandle (Movie mMovie)`

`GetMoviePosterPict retrieves a movie's poster frame in the QuickTime for Windows picture format.`

Parameters *Movie* mMovie

The movie object.

Return A picture object. A NULL return indicates failure. You can also use GetMoviesError and GetMoviesStickyError to test for failure of this call.

Comments This function may be called whether a movie is playing or not. The picture object returned is unusable by Windows directly. Use the function PictureToDIB to convert it to a Windows Device Independent Bitmap (DIB). An alternative to converting the image is using DrawPicture to display it at specified coordinates.

Example

```
Movie mMovie;
PicHandle phPicture;
RECT rcPicture;
HDC hdc;
•
•
// Retrieve Poster Frame, then display it on the screen

   if ((phPicture = GetMoviePosterPict (mMovie)) != NULL)
      DrawPicture
 (hdc, phPicture, &rcPicture, NULL);
```

See Also:

Functions DrawPicture, GetMoviePict, GetMoviePosterTime, PictureToDIB, GetMoviesError, GetMoviesStickyError

## *GetMoviePosterTime*

Syntax `TimeValue GetMoviePosterTime (Movie mMovie)`

`GetMoviePosterTime` finds the poster's time in the movie.

Parameters *Movie* mMovie

The movie object.

Return The `TimeValue` of the poster frame. You can use GetMoviesError and GetMoviesStickyError to test for failure of this call.

Comments The poster is an image from the movie which may be used to characterize it when the movie is not running. For example, the poster might serve as a visual representation of a movie's contents in an open dialog. To get the poster picture object itself use GetMoviePosterPict.

Example

```
TimeValue tvPoster;
Movie mMovie;
•
•
tvPoster = GetMoviePosterTime (mMovie);
```

See Also:

Functions ConvertTimeScale, GetMovieDuration, GetMoviesError, GetMoviePosterPict, GetMoviesStickyError, MCGetCurrentTime

Data Types `TimeValue`

## *GetMoviePreferredRate*

Syntax `LFIXED GetMoviePreferredRate (Movie mMovie)`

`GetMoviePreferredRate` determines the preferred rate at which a movie is played.

Parameters *`Movie`* `mMovie`

The movie object.

Return An LFIXED value which is the preferred rate of the movie expressed as a multiplier of the recorded rate. For example, a return value of 1.0 means play the movie at the recorded rate. A return value of 1.5 would mean play the movie 1.5 times faster than its recorded rate. Use GetMoviesError and GetMoviesStickyError to test for failure of this call.

Comments The return value can be passed on to MCDoAction mcActionPlay to play the movie at the preferred rate.

Example

```
Movie mMovie;
MovieController mcController;
LFIXED  lfxRate;
•
•
// Play the movie at the preferred rate

   lfxRate = GetMoviePreferredRate (mMovie);
   MCDoAction  (mcController, mcActionPlay, (LPVOID) lfxRate);
```

See Also:

Functions GetMoviePreferredVolume, GetMoviesError, GetMoviesStickyError

MCDoAction mcActionPlay

## *GetMoviePreferredVolume*

Syntax `SFIXED GetMoviePreferredVolume (Movie mMovie)`

`GetMoviePreferredVolume` returns a movie's preferred volume setting.

Parameters *`Movie`* `mMovie`

The movie object.

Return — An SFIXED value ranging from 256 to -256. Negative values represent volume levels that play no sound but preserve the absolute value of the volume setting. Use GetMoviesError and GetMoviesStickyError to test for failure of this call.

Comments — The return value can be passed on to MCDoAction using the action mcActionSetVolume to play the movie at the preferred volume.

Example

```
Movie mMovie;
MovieController mcController;
SFIXED sfxVolume;
•
•
// Set the volume to the preferred level

   sfxVolume = GetMoviePreferredVolume (mMovie);
   MCDoAction (mcController, mcActionSetVolume,
      (LPVOID) sfxVolume);
```

See Also:

Functions — GetMoviePreferredRate, GetMoviesError, GetMoviesStickyError

MCDoAction — mcActionSetVolume

## *GetMoviesError*

Syntax

```
OSErr GetMoviesError (VOID)
```

`GetMoviesError` retrieves the current QuickTime for Windows movie error value and resets it to 0.

Parameters — This routine takes no parameters.

Return — The result code from the previous eligible QuickTime for Windows call. Eligible QuickTime for Windows calls are calls that operate on movies (as opposed to movie controllers) and require a movie object.

Comments — Use this call to obtain the result code for QuickTime for Windows movie calls that do not return an error as a function result. Even if a movie routine explicitly returns an error as a function result, the result is also available using the `GetMoviesError` function. See Appendix A for error condition values.

Example

```
Movie mMovie;
LFIXED lfxRate;
•
•
lfxRate = GetMoviePreferredRate (mMovie);
if (GetMoviesError())
    {
```

```
    MessageBox (NULL, "GetMoviePreferredRate Failure",
      "Program", MB_OK);
    }
```

See Also:

Functions GetMoviesStickyError, ClearMoviesStickyError

Data Types OSErr

### *GetMoviesStickyError*

Syntax

```
OSErr GetMoviesStickyError (VOID)
```

GetMoviesStickyError retrieves the sticky error value. The sticky error value is the first non-zero result code returned by an eligible QuickTime for Windows routine since ClearMoviesStickyError was last called.

Parameters This routine takes no parameters.

Return The first non-zero result code from the previous eligible QuickTime for Windows calls since the sticky error value was last cleared. Eligible QuickTime for Windows calls operate on movies (as opposed to movie controllers) and require a movie object.

Comments Even if a movie routine explicitly returns an OSErr, the result is also available using the GetMoviesStickyError function.

The GetMoviesStickyError function does not clear the sticky error value. Use the ClearMoviesStickyError function for this purpose.

A result code will not be placed into the sticky error value until the field has been cleared. Your application should clear the sticky error value to ensure that it does not contain a stale result code.

Example

```
Movie mMovie;
LFIXED lfxRate;
•
•
// Assume call produces an error code

   lfxRate = GetMoviePreferredRate (mMovie);

// Assume other calls follow with no errors
•
•
   if (GetMoviesStickyError())
      {
      MessageBox (NULL, "GetMoviePreferredRate Failure",
         "Program", MB_OK);
      ClearMoviesStickyError();
      }
```

See Also:

Functions GetMoviesError, ClearMoviesStickyError

## *GetMovieStatus*

Syntax `OSErr GetMovieStatus (Movie mMovie, LPVOID lpvReserved)`

`GetMovieStatus` looks for defects in a movie and returns a defined error condition if any are found.

Parameters *`Movie`* `mMovie`

The movie object.

*`LPVOID`* `lpvReserved`

Reserved.

Return Zero if no error condition. Non-zero if error condition. See Appendix A for error condition values. You can use GetMoviesError and GetMoviesStickyError to test for failure of this call.

Comments Inconsistencies found in the movie data are reported.

Example

```
MovieFile mfMovie;
Movie mMovie;
LPVOID lpvReserved;
•
•
OpenMovieFile ("SAMPLE.MOV", &mfMovie, OF_READ);
NewMovieFromFile (&mMovie, mfMovie, NULL, NULL, 0, NULL);
CloseMovieFile (mfMovie);
if (GetMovieStatus (mMovie, lpvReserved))
   {
   /* Display error message. */
   }
```

See Also:

Functions GetMoviesError, GetMoviesStickyError

## *GetMovieTime*

Syntax `TimeValue GetMovieTime (Movie mMovie,` TimeRecord `FAR *trRecord)`

`GetMovieTime` retrieves the current time of a movie at the point that the routine is called.

Parameters *`Movie`* `mMovie`

The movie object.

TimeRecord `*trRecord`

The address of a TimeRecord which will be filled with the movie's time scale, time base and current time. The high 32 bits of the time value field are always 0, while the low 32 bits represent the same value as the returned `TimeValue`.

Return

A `TimeValue` containing the movie's current time at the point the routine is called. Use GetMoviesError and GetMoviesStickyError to test for failure of this call.

Comments

A movie's time coordinate system is based on a time scale scored in time units. The number of units that pass per second quantifies the scale: a time scale of 26 means that 26 units pass per second and each time unit is 1/26 of a second.

When the duration of all or part of a movie is needed, it is expressed as the length of the portion of the movie in the number of time units it contains. Particular points in a movie can be identified by a time value, which is the number of time units to that point from the beginning of the movie.

Different movies may have different time scales. Use ConvertTimeScale to compare `TimeValues` between different movies.

Example

```
Movie mMovie;
TimeValue tvCurrentTime;
TimeRecord trTimeData;
•
•
// Get the movie's current time

   tvCurrentTime = GetMovieTime (mMovie, &trTimeData);
```

See Also:

Functions

ConvertTimeScale, GetMovieDuration, MCGetCurrentTime, GetMovieTimeScale, GetMoviesError, GetMoviesStickyError

Data Types

`TimeScale, TimeValue`

## *GetMovieTimeScale*

Syntax

```
TimeScale GetMovieTimeScale (Movie mMovie)
```

`GetMovieTimeScale` retrieves the time scale of a movie.

Parameters

*`Movie`* `mMovie`

The movie object.

Return

The time scale of the movie, i.e. the number of time units that pass per second. Use GetMoviesError and GetMoviesStickyError to test for failure of this call.

Comments

A movie's time coordinate system is based on a time scale scored in time units. The number of units that pass per second quantifies the scale: a time scale of 26 means that 26 units pass per second and each time unit is 1/26 of a second.

When the duration of all or part of a movie is needed, it is expressed as the length of the portion of the movie in the number of time units it contains. Particular points in a movie can be identified by a time value, which is the number of time units to that point from the beginning of the movie.

Different movies may have different time scales. Use ConvertTimeScale to compare `TimeValues` between different movies.

Example

```
LONG lSize;
Movie mMovie;
TimeValue tvStart, tvDuration;
MovieController mcController;
•
•
// Get the number of bytes from the current position to two
// seconds later

   tvStart = MCGetCurrentTime(mcController, NULL);
   tvDuration = 2 * GetMovieTimeScale (mMovie);
   lSize = GetMovieDataSize (mMovie, tvStart, tvDuration);
```

See Also:

Functions ConvertTimeScale, GetMovieDuration, MCGetCurrentTime, GetMoviesError, GetMoviesStickyError

Data Types `TimeScale, TimeValue`

## *GetMovieUserData*

Syntax `UserData GetMovieUserData (Movie mMovie)`

`GetMovieUserData` retrieves a handle to a list of user data belonging to a movie. This handle is maintained internally by QuickTime for Windows. You do not need to free it when you are finished using it.

Parameters *`Movie`* `mMovie`

The movie object.

Return The handle to a list of user data. You can use GetMoviesError and GetMoviesStickyError to test for failure of this call.

Comments A movie's user data list is placed in a movie by its creator and may contain items of various types. A common type is text containing copyright data, names of people involved in the movie's production, special hardware and software requirements, and other types of information about the movie. By convention, text user data types start with a "©" symbol. A list of commonly used text user data types may be found in Section 16 of the overview.

Example See the example in the description of GetUserDataText.

See Also:

Functions CountUserDataType, GetNextUserDataType, GetUserData, GetUserDataText, GetMoviesError, GetMoviesStickyError

Data Types UserData

### *GetNextUserDataType*

Syntax `OSType GetNextUserDataType (UserData udData, OSType ostType)`

This function is used to retrieve the next user data type in a user data list.

Parameters *UserData* udData

The handle to the user data list.

*OSType* ostType

The user data type. If zero is used, the first user data type in the list is returned. If a user data type is used, the next user data type is returned.

Return The next user data type, or zero if no more types are present. You can use GetMoviesError and GetMoviesStickyError to test for failure of this call.

Comments A movie's user data list is placed in a movie by its creator and may contain items of various types. A common type is text containing copyright data, names of people involved in the movie's production, special hardware and software requirements, and other types of information about the movie. By convention, text user data types start with a "©" symbol. A list of commonly used text user data types may be found in Section 16 of the overview.

Example See the example in the description of GetUserDataText.

See Also:

Functions CountUserDataType, GetUserData, GetUserDataText, GetMoviesError, GetMoviesStickyError

Data Types UserData

### *GetPictureFileHeader*

Syntax `OSErr GetPictureFileHeader (PicFile pfPicture, LPRECT lprcFrame, OpenCPicParams FAR *lpocppHeader)`

`GetPictureFileHeader` retrieves the header to the picture file and the picture frame rectangle.

Parameters *PicFile* pfPicture

The picture file reference value returned by OpenPictureFile.

*LPRECT* lprcFrame

The address of the picture frame rectangle.

OpenCPicParams *FAR* *lpocppHeader

The address of the picture file header data.

Return Zero if no error condition. Non-zero if error condition. See Appendix A for error condition values. The picture frame rectangle and picture file header referenced by the second and third parameters are populated with the retrieved data. You can use GetMoviesError and GetMoviesStickyError to test for failure of this call.

Comments Picture files are characterized by the DOS file suffix ".PIC". They are DOS versions of Macintosh PICT and JFIF files.

*Note: All QuickTime for Windows routines referencing a RECT or POINT assume client device coordinates.*

Example

```
PicFile pfPicture;
OpenCPicParams  ocppHeader;
RECT rcFrame;
•
•
OpenPictureFile ("HOUSE.PIC", &pfPicture, OF_READ);
GetPictureFileHeader (pfPicture, &rcFrame, &ocppHeader);
ClosePictureFile (pfPicture);
```

See Also:

Functions ClosePictureFile, DrawPictureFile, GetPictureFileInfo, GetPictureInfo, GetMoviesError, GetMoviesStickyError, OpenPictureFile

Data Types OpenCPicParams

## *GetPictureFileInfo*

Syntax `OSErr GetPictureFileInfo (PicFile pfPicture, ImageDescription FAR *idImageInfo)`

`GetPictureFileInfo` retrieves detailed information about a picture file.

Parameters *PicFile pfPicture*

The picture file reference value referred to by OpenPictureFile.

ImageDescription *FAR* *idImageInfo

The address of the image descriptor.

Return — Zero if no error condition. Non-zero if error condition. See Appendix A for error condition values. The image descriptor record is populated with information on the picture file. You can use GetMoviesError and GetMoviesStickyError to test for failure of this call.

Comments — The information retrieved by `GetPictureFileInfo` is more detailed than that retrieved by GetPictureFileHeader. Picture files are characterized by the DOS file suffix ".PIC". They are DOS versions of Macintosh PICT and JFIF files.

Example

```
PicFile pfPicture;
ImageDescription  idImageInfo;
•
•
OpenPictureFile ("HOUSE.PIC", &pfPicture, OF_READ);
idImageInfo.idSize = sizeof (ImageDescription );
GetPictureFileInfo (pfPicture, &idImageInfo);
ClosePictureFile (pfPicture);
```

See Also:

Functions — ClosePictureFile, GetPictureFileHeader, GetPictureInfo, GetMoviesError, GetMoviesStickyError, `OpenPictureFile`

Data Types — ImageDescription

### *GetPictureFromFile*

Syntax — `PicHandle GetPictureFromFile (PicFile pfPicture)`

`GetPictureFromFile` extracts a picture from a picture file.

Parameters — *`PicFile`* `pfPicture`

The reference value assigned by OpenPictureFile.

Return — A `PicHandle` for subsequently referencing the picture, `NULL` if failure. You can also use GetMoviesError and GetMoviesStickyError to test for failure of this call.

Comments — You can use the picture object returned by `GetPictureFromFile` to create a Windows Device Independent Bitmap (DIB).

Example

```
PicFile pfPicture;
PicHandle phThePict;
•
•
if (!OpenPictureFile ("PICTURE.PIC", &pfPicture, OF_READ))
   {
   phThePict = GetPictureFromFile (pfPicture);
   ClosePictureFile (pfPicture);
   }
```

See Also:

Functions OpenPictureFile, ClosePictureFile, GetMoviesError, GetMoviesStickyError

## *GetPictureInfo*

Syntax `OSErr GetPictureInfo (PicHandle, ImageDescription FAR *)`

`GetPictureInfo` retrieves detailed information about an image.

Parameters *`PicHandle phThePict`*

The picture object.

ImageDescription *`FAR *idImageInfo`*

The address of the image descriptor.

Return Zero if no error condition. Non-zero if error condition. See Appendix A for error condition values. The image descriptor record referenced by the second parameter is populated with information about the image. You can use GetMoviesError and GetMoviesStickyError to test for failure of this call.

Comments Pictures are created using GetMoviePict, GetMoviePosterPict and GetPictureFromFile. *Note: this routine is a limited version of its Macintosh counterpart, in that only information about the first picture is retrieved. Future releases of QuickTime for Windows will upgrade this function.*

Example

```
Movie mMovie;
PicHandle phThePict;
ImageDescription idImageInfo;
•
•
if ((phThePict = GetMoviePosterPict (mMovie)) != NULL)
   {
   idImageInfo.idSize = sizeof (ImageDescription);
   GetPictureInfo (phThePict, &idImageInfo);
   }
```

See Also:

Functions GetPictureFileHeader, GetPictureFileInfo, GetMoviePict, GetMoviePosterPict, GetMoviesError, GetMoviesStickyError

Data Types ImageDescription

## *GetPicturePalette*

Syntax `HPALETTE GetPicturePalette (PicHandle phThePict)`

`GetPicturePalette` retrieves a palette from a picture.

Parameters *`PicHandle`* `phThePict`

A picture object.

Return — A handle to the picture's palette, `NULL` if the picture has no palette. You can use GetMoviesError and GetMoviesStickyError to test for failure of this call.

Comments — The returned `HPALETTE` can be used to display pictures using a Windows palette. You must free it, when you are done with it, using `DeleteObject`.

`GetPicturePalette` always attempts to return a palette. If the picture does not have one, it returns a default palette.

Example

```
PicFile pfPicture;
PicHandle phThePict;
HPALETTE hPal;
•
•
if (!OpenPictureFile ("PICTURE.PIC", &pfPicture, OF_READ))
   {
   phThePict = GetPictureFromFile (pfPicture);
   hPal = GetPicturePalette (phThePict);
   ClosePictureFile (pfPicture);
   }
```

See Also:

Functions — ClosePictureFile, GetMoviesError, OpenPictureFile, GetMoviesStickyError, GetPictureFromFile

## *GetSoundInfo*

Syntax — `OSErr GetSoundInfo (Movie, SoundDescription FAR *)`

`GetSoundInfo` retrieves information about a movie's sound.

Parameters — *`Movie`* `mMovie`

The movie object.

SoundDescription *`FAR`* `*sdSoundInfo`

The address of the sound description data.

Return — Zero if no error condition. Non-zero if error condition. See Appendix A for error condition values. The sound description record is populated with data about the movie's sound. You can use the routines GetMoviesError and GetMoviesStickyError to test for failure of this call.

Comments — `GetSoundInfo` retrieves useful information about a movie's sound, such as number of channels, sample size and sampling rate. *Note: this routine is a limited version of its Macintosh counterpart, in that only information about the first track is retrieved (*tracks *are not meaningful under QuickTime for Windows). Future releases of QuickTime for Windows will upgrade this function.*

Example

```
Movie mMovie;
SoundDescription sdSoundInfo;
•
•
sdSoundInfo.descSize = sizeof (SoundDescription);
GetSoundInfo (mMovie, &sdSoundInfo);
if ((SHORT) sdSoundInfo.numChannels == 1)
   {
   /* Tell user sound is mono. */
   }
```

See Also:

Functions GetVideoInfo, GetMoviesError, GetMoviesStickyError

Data Types SoundDescription

## *GetUserData*

Syntax

```
OSErr GetUserData (UserData udData, LPHANDLE lphData, OSType
ostType, LONG lIndex, LPLONG lplSize)
```

`GetUserData` retrieves data from an item in a user data list.

Parameters

*`UserData`* `udData`

The handle to the user data list.

*`LPHANDLE`* `lphData`

A handle for a block memory that will receive the requested data. This function will reallocate this memory to accommodate the data, if necessary.

*`OSType`* `ostType`

The user data type.

*`LONG`* `lIndex`

Each user data item is identified by a unique index value. Index values are assigned sequentially within a user data type starting with 1.

*`LPLONG`* `lplSize`

The size of the data returned.

Return

Zero if no error condition. Non-zero if error condition. See Appendix A for error condition values. You can use the routines GetMoviesError and GetMoviesStickyError to test for failure of this call.

Comments

A movie's user data list is placed in a movie by its creator and may contain items of various types. A common type is text containing copyright data, names of people

involved in the movie's production, special hardware and software requirements, and other types of information about the movie. By convention, text user data types start with a "©" symbol. A list of commonly used text user data types may be found in Section 16 of the overview.

Example See the example in the description of GetUserDataText.

See Also:

Functions CountUserDataType, GetMovieUserData, GetUserDataText, GetNextUserDataType, GetMoviesError, GetMoviesStickyError

Data Types `UserData`

### *GetUserDataText*

Syntax

```
OSErr GetUserDataText (UserData udData, LPHANDLE lphData, OSType
ostType, LONG lIndex, UINT uRegionTag, LPLONG lplSize)
```

`GetUserDataText` retrieves text from an item in a user data list. Each user data text item may have alternative text. For example, multiple languages may be supported. Each alternative text value is identified by a region code. A table of these codes is provided in Appendix B.

Parameters

*`UserData`* `udData`

The handle to the user data list.

*`LPHANDLE`* `lphData`

A handle for a block memory that will receive the requested data. This function will reallocate this memory to accommodate the data, if necessary.

*`OSType`* `ostType`

The user data type.

*`LONG`* `lIndex`

Each user data item is identified by a unique index value. Index values are assigned sequentially within a user data type starting with 1.

*`UINT`* `uRegionTag`

A region tag that may identify alternate text. A table of these codes is provided in Appendix B.

*`LPLONG`* `lplSize`

The size of the text value returned.

Return

Zero if no error condition. Non-zero if error condition. See Appendix A for error condition values. You can use the routines GetMoviesError and GetMoviesStickyError to test for failure of this call.

Comments

A movie's user data list is placed in a movie by its creator and may contain items of various types. A common type is text containing copyright data, names of people involved in the movie's production, special hardware and software requirements, and other types of information about the movie. By convention, text user data types start with a "©" symbol. A list of commonly used text user data types may be found in Section 16 of the overview.

Example

```
// A function that steps through the user data list

void CheckUserDataFunctions (Movie mCheck, UINT uRegionTag)
   {
   UserData udMovie;
   OSType   osType;
   LONG     lUserDataCount;
   LONG     i;
   LONG     lByteCount;
   HGLOBAL  hgMem;
   char     szText [256];
   LPSTR    lpszText;

// Get the user data handle

   udMovie = GetMovieUserData (mCheck);

// Allocate memory - note 128 is arbitrary amount

   hgMem = GlobalAlloc (GMEM_MOVEABLE, 128);

// Find the first user data type

   osType = GetNextUserDataType (udMovie, 0);

// Parse the user data list

   while ( osType != 0)
      {
      lUserDataCount = CountUserDataType (udMovie, osType);
      for ( i = 1; i <= lUserDataCount; i++)
         {
         if (GetUserDataText (udMovie, &hgMem, osType, i,
            uRegionTag, &lByteCount) == 0)
            {
            lpszText = (LPSTR) GlobalLock (hgMem);
            lpszText [lByteCount] = '\0';
            /* Display the text. */
            wsprintf (szText, "User Data of Type: %ld/%ld/%s",
               osType, i, lpszText);
            GlobalUnlock (hgMem);
            }
         }
         osType = GetNextUserDataType (udMovie, osType);
```

```
        }

// The program must free the memory

    GlobalFree (hgMem);
    }
```

See Also:

Functions CountUserDataType, GetMovieUserData, GetUserData, GetNextUserDataType, GetMoviesError , GetMoviesStickyError

Data Types UserData, OSType

## *GetVideoInfo*

Syntax `OSErr GetVideoInfo (Movie mMovie, ImageDescription FAR *)`

`GetVideoInfo` retrieves information about a movie's video.

Parameters *`Movie`* `mMovie`

The movie object.

ImageDescription *`FAR`* `*idVideoInfo`

The address of the image description data.

Return Zero if no error condition. Non-zero if error condition. See Appendix A for error condition values. The image description data is populated with information about the movie's video data. Use GetMoviesError and GetMoviesStickyError to test for failure of this call.

Comments `GetVideoInfo` retrieves useful information about a movie's video, such as horizontal and vertical resolution, frame count and compressor type.

*Note: this routine is a limited version of its Macintosh counterpart, in that only information about the first track is retrieved (*tracks *are not meaningful under QuickTime for Windows). Future releases of QuickTime for Windows will upgrade this function.*

Example

```
Movie mMovie;
PicHandle phThePict;
ImageDescription idVideoInfo;
•
•
idVideoInfo.idSize = sizeof (ImageDescription);
GetVideoInfo (mMovie, &idVideoInfo);
```

See Also:

Functions GetSoundInfo, GetPictureInfo, GetMoviesError, GetMoviesStickyError

Data Types ImageDescription

## *KillPicture*

Syntax `VOID KillPicture (PicHandle phPicture)`

`KillPicture` frees any memory being used by a QuickTime for Windows picture. Your program should call this routine when it is done working with a QuickTime for Windows picture.

Parameters *`PicHandle`* `phPicture`

The picture object whose memory is being released.

Return None. Use GetMoviesError and GetMoviesStickyError to test for failure of this call.

Comments Either `KillPicture` or DisposePicture must be called, ultimately, for each picture instantiated by your program. It does not affect the DOS file containing the picture.

Example

```
PicHandle phPicture;
PicFile pfPicture;
•
•
if (!OpenPictureFile ("PICTURE.PIC", &pfPicture, OF_READ))
   {
   phPicture = GetPictureFromFile (pfPicture);
   ClosePictureFile (pfPicture);
   }
•
•
KillPicture (phPicture);
```

See Also:

Functions GetPictureFromFile, OpenPictureFile, ClosePictureFile, DisposePicture, GetMoviesError, GetMoviesStickyError

## *MAKELFIXED*

Syntax `MAKELFIXED(integer, fract)`

`MAKELFIXED` is a macro used to construct an LFIXED variable.

Parameters *`integer`*

A signed sixteen-bit value representing the integral part of the LFIXED variable.

*`fract`*

An unsigned sixteen-bit value representing the fractional part of the LFIXED variable.

Comments LFIXED variables are normally used to hold movie rates in QuickTime for Windows. For example, the LFIXED value `0x0028000` could be used to represent a rate of 2.5.

Example

```
LFIXED lfxRate;

// Set the movie rate to 2.5

   lfxRate = MAKELFIXED(0x0002, 0x8000);
```

See Also:

Functions MAKESFIXED (macro)

Data Types LFIXED, SFIXED

## *MAKESFIXED*

Syntax `MAKESFIXED(integer, fract)`

`MAKESFIXED` is a macro used to construct an SFIXED variable.

Parameters *integer*

A signed eight-bit value representing the integral part of the SFIXED variable.

*fract*

An unsigned eight-bit value representing the fractional part of the SFIXED variable.

Comments SFIXED variables are normally used to hold movie sound track volumes in QuickTime for Windows. For example, the SFIXED value `0x0080` could be used to represent a sound volume of 0.5.

Example

```
SFIXED sfxVolume;

// Set the movie sound volume to 0.5

   sfxVolume = MAKESFIXED(0x00, 0x08);
```

See Also:

Functions MAKELFIXED (macro)

Data Types LFIXED, SFIXED

## *MCActionFilter*

Syntax

```
BOOL CALLBACK MCActionFilter (MovieController mcController, UINT
uAction, LPVOID lpParam, LONG lRefCon)
```

`MCActionFilter` is the prototype for the filter function set by the routine

MCSetActionFilter. It shows the parameters you must pass to your filter, and the value your filter must return.

Parameters

*MovieController* mcController

The movie controller object.

*UINT* uAction

The action to be filtered, which is the same as the one passed to MCDoAction.

*LPVOID* lpParam

The optional extra parameter that modifies the action referenced by uAction, which is the same as the one passed to MCDoAction.

*LONG* lRefcon

Additional data of use to the filter when processing the action. Should be coded as 0L if not used.

Return

TRUE indicates that the movie controller doesn't have to handle the action (since your filter has taken appropriate action), FALSE that it does.

Comments

MCActionFilter is not a defined QuickTime for Windows function. It is a prototype only, used as a template for your filter functions.

Example

```
BOOL CALLBACK __export MyFilter (MovieController, UINT,
   LPVOID, LONG);
•
•
BOOL CALLBACK __export MyFilter (MovieController mcController,
   UINT uAction, LPVOID lpVoid, LONG lRefCon)
   {
   switch (uAction)
      {
      /* cases */
      }

   return FALSE;
   }
```

See Also:

Functions

MCSetActionFilter

## *MCActivate*

Syntax

```
ComponentResult MCActivate (MovieController mcController, HWND
hWnd, BOOL bActivate)
```

MCActivate sets a movie controller's state to active or inactive.

Parameters

*MovieController* mcController

The movie controller object.

*HWND* hWnd

The controller parent's window handle.

*BOOL* bActivate

TRUE to set the controller active. FALSE to set the controller inactive.

Return

Zero if no error condition. Non-zero if error condition. See Appendix A for error condition values.

Comments

An inactive movie controller cannot receive mouse clicks and its appearance is grayed. It can still receive keyboard input, if the keyboard interface is active. Movie controllers are created with an active state by default.

A movie/movie controller pair can have opposing states. For example, a playing movie's controller can be deactivated, graying it and prohibiting further mouse input, but the movie will keep playing. In the case where the controller is active and the movie is inactive, the movie will receive no service from the QuickTime for Windows scheduler and will not play even though the controller is functional.

More than one movie controller can be active at a time. Both attached and detached movie controllers can be made inactive.

There is no QuickTime for Windows function to query the active state of a movie controller.

Example

```
MovieController mcController;
HWND hWndParent;
•
•
// Make the controller inactive to prevent its use

   MCActivate (mcController, hWndParent, FALSE);
```

See Also:

Functions

GetMovieActive, SetMovieActive

MCDoAction

mcActionActivate

## *MCDoAction*

Syntax

```
ComponentResult MCDoAction(MovieController mcController, UINT
uAction, LPVOID lpvParams)
```

MCDoAction causes a movie controller perform a specified action, based on the parameters passed to it.

Parameters

*MovieController* mcController

The movie controller object.

*UINT* uAction

An action flag parameter with the prefix "mcAction...". Each action flag parameter is documented in detail in the following pages.

*LPVOID* lpvParams

A modifier of the uAction parameter.

Return

Zero if no error condition. Non-zero if error condition. See Appendix A for error condition values.

Comments

MCDoAction is a powerful and versatile routine, often called by QuickTime for Windows internally, that is used to dictate most of the movie controller's behavior by taking particular defined actions. There are many mcActions in the QuickTime for Windows API, ranging from starting the movie to toggling low-level attributes. In most cases, an additional parameter is required to modify the task of the mcAction parameter. Often this is a boolean value which can turn a certain attribute on or off, or a pointer to a value holding state information.

For example, your application might define a menu item that stops all currently playing movies. When the user selects this menu item, your application could use the MCDoAction function to instruct each controller to stop playing. You would do so by specifying the mcActionPlay action with the last parameter set to specify that the controller stop playing the movie.

Often you will issue a MCDoAction call in response to a user action, such as a menu selection. More importantly, you can trap a MCDoAction event issued by QuickTime for Windows itself in a filter, since QuickTime for Windows passes all MCDoAction calls through your filter (if you have one) before processing them. For further details, see MCSetActionFilter.

Example

```
MovieController mcController;
•
•
// Disable the keyboard interface

   MCDoAction (mcController, mcActionSetKeysEnabled,
      (LPVOID) FALSE);
```

See Also:

Functions

MCSetActionFilter

## *MCDoAction* mcActionActivate

Syntax

```
ComponentResult MCDoAction (MovieController mcController, UINT
uAction, LPVOID lpvParams)
```

MCDoAction with the mcActionActivate parameter causes the movie controller to

be activated.

Parameters

*MovieController* mcController

The movie controller object.

*UINT* uAction

mcActionActivate

*LPVOID* lpvParams

NULL

Return

Zero if no error condition. Non-zero if error condition. See Appendix A for error condition values.

Comments

An inactive movie controller cannot receive mouse clicks and its appearance is grayed. It can still receive keyboard input, if the keyboard interface is active. Movie controllers are created with an active state by default.

A movie/movie controller pair can have opposing states. For example, a playing movie's controller can be deactivated, graying it and prohibiting further mouse input, but the movie will keep playing. In the case where the controller is active and the movie is inactive, the movie will receive no service from the QuickTime for Windows scheduler and will not play even though the controller is functional.

More than one movie controller can be active at a time. Both attached and detached movie controllers can be made inactive.

Example

```
MovieController mcController;
•
•
// Activate the movie controller

   MCDoAction (mcController, mcActionActivate, NULL);
```

See Also:

Functions

MCActivate, MCDoAction, MCSetActionFilter

MCDoAction

mcActionDeactivate

## *MCDoAction* mcActionBadgeClick

Syntax

```
ComponentResult MCDoAction (MovieController  mcController, UINT
uAction, LPVOID lpvParams)
```

Your filter receives a mcActionBadgeClick notification when the user has clicked on a movie's badge.

Parameters

*MovieController* mcController

The movie controller object.

*UINT* uAction

mcActionBadgeClick

*LPVOID* lpvParams

NULL

Return — Zero if no error condition. Non-zero if error condition. See Appendix A for error condition values.

Comments — Your application should normally never issue this action. An action filter function may trap it when the user has clicked on a movie's badge. See the description of MCSetActionFilter for details on the filter procedure.

If a controller's badge capability is enabled, then the badge is displayed whenever the controller is not visible. When the controller is visible, the badge is not displayed. If the badge capability is disabled, the badge is never displayed.

Example — See the sample program listing FILTERS.C in the QuickTime for Windows Tutorial section for further information about filters.

See Also:

Functions — MCDoAction, MCSetActionFilter

MCDoAction — mcActionGetUseBadge

### *MCDoAction mcActionControllerSizeChanged*

Syntax

```
ComponentResult MCDoAction(MovieController mcController, UINT
uAction, LPVOID lpvParams)
```

Your filter receives a `mcActionControllerSizeChanged` notification when the user has resized the movie controller.

Parameters — *MovieController* mcController

The movie controller object.

*UINT* uAction

mcActionControllerSizeChanged

*LPVOID* lpvParams

NULL

Return — Zero if no error condition. Non-zero if error condition. See Appendix A for error

condition values.

Comments — Your application should normally never issue this action. An action filter function may trap it when the user has resized the movie controller. See the description of MCSetActionFilter for details on the filter procedure.

Example — See the sample program listing FILTERS.C in the QuickTime for Windows Tutorial section for further information about filters.

See Also:

Functions — MCDoAction, MCSetActionFilter

## *MCDoAction* mcActionDeactivate

Syntax

```
ComponentResult MCDoAction(MovieController mcController, UINT
uAction, LPVOID lpvParams)
```

MCDoAction with the `mcActionDeactivate` parameter causes the movie controller to be deactivated.

Parameters

*MovieController* mcController

The movie controller object.

*UINT* uAction

mcActionDeactivate

*LPVOID* lpvParams

NULL

Return — Zero if no error condition. Non-zero if error condition. See Appendix A for error condition values.

Comments — An inactive movie controller cannot receive mouse clicks and its appearance is grayed. It can still receive keyboard input, if the keyboard interface is active. Movie controllers are created with an active state by default.

A movie/movie controller pair can have opposing states. For example, a playing movie's controller can be deactivated, graying it and prohibiting further mouse input, but the movie will keep playing. In the case where the controller is active and the movie is inactive, the movie will receive no service from the QuickTime for Windows scheduler and will not play even if the controller is functional.

More than one movie controller can be active at a time. Both attached and detached movie controllers can be made inactive.

Example

```
MovieController mcController;
•
•
```

```
// Deactivate the movie controller

   MCDoAction (mcController, mcActionDeactivate, NULL);
```

See Also:

Functions MCActivate, MCDoAction, MCSetActionFilter

MCDoAction mcActionActivate

## *MCDoAction* mcActionDraw

Syntax `ComponentResult MCDoAction (MovieController mcController, UINT uAction, LPVOID lpvParams)`

MCDoAction with the `mcActionDraw` parameter causes the movie image to be redrawn.

Parameters *`MovieController`* `mcController`

The movie controller object.

*`UINT`* `uAction`

`mcActionDraw`

*`LPVOID`* `lpvParams`

NULL

Return Zero if no error condition. Non-zero if error condition. See Appendix A for error condition values.

Comments Your application can use MCDoAction with this parameter to send an update event to a movie controller.

Example

```
MovieController mcController;
•
•
// Update the movie image

   MCDoAction (mcController, mcActionDraw, NULL);
```

See Also:

Functions MCDoAction, MCDraw, MCSetActionFilter

## *MCDoAction* mcActionGetFlags

Syntax `ComponentResult MCDoAction(MovieController mcController, UINT uAction, LPVOID lpvParams)`

MCDoAction with the `mcActionGetFlags` parameter retrieves a set of flag values that determine the behavior of the Movie Controller.

Parameters

*MovieController* mcController

The movie controller object.

*UINT* uAction

mcActionGetFlags

*LPVOID* lpvParams

A pointer to a long integer that contains the set of flag values:

- mcFlagsUseWindowPalette
- mcFlagSuppressStepButtons
- mcFlagSuppressSpeakerButton

Return

Zero if no error condition. Non-zero if error condition. See Appendix A for error condition values.

Comments

The retrieved flags are defined as follows:

mcFlagSuppressStepButtons - Determines whether the movie controller displays the step buttons. The step buttons allow the user to step the movie forward or backward one frame at a time. If this flag is set, the controller does not display the step buttons.

mcFlagSuppressSpeakerButton - Determines whether the movie controller displays the speaker button. The speaker button allows the user to control the movie's sound. If this flag is set, the controller does not display the speaker button.

mcFlagsUseWindowPalette - Determines whether the movie controller constructs a custom color palette, based on the color values found in the movie. This flag only works with display drivers that support palettes, typically those drivers that handle colors at pixel depth eight.

Example

```
MovieController mcController;
LONG lFlags;
•
•
// Hide the speaker button

MCDoAction (mcController, mcActionGetFlags, (LPVOID) &lFlags);
lFlags |= mcFlagSuppressSpeakerButton;
MCDoAction (mcController, mcActionSetFlags, (LPVOID) lFlags);
```

See Also:

Functions

MCDoAction, MCSetActionFilter

MCDoAction

mcActionSetFlags

## *MCDoAction* mcActionGetKeysEnabled

Syntax

```
ComponentResult MCDoAction (MovieController mcController, UINT
uAction, LPVOID lpvParams)
```

MCDoAction with the `mcActionGetKeysEnabled` parameter determines whether a movie controller's keyboard interface is enabled.

Parameters

*`MovieController`* `mcController`

The movie controller object.

*`UINT`* `uAction`

`mcActionGetKeysEnabled`

*`LPVOID`* `lpvParams`

A pointer to a boolean, set to `TRUE` if keyboard interface is enabled, `FALSE` if not.

Return

Zero if no error condition. Non-zero if error condition. See Appendix A for error condition values.

Comments

An inactive keyboard interface is the default attribute for a new movie controller. The keyboard interface will remain in an active state even if the controller is set inactive (i.e. grayed and unable to receive mouse clicks).

Example

```
MovieController mcController;
BOOL bEnabled;
•
•
// Enable keystrokes for movie if they're disabled

   MCDoAction (mcController, mcActionGetKeysEnabled, (LPVOID)
&bEnabled);
   if (!bEnabled)
      MCDoAction (mcController, mcActionSetKeysEnabled, LPVOID)
TRUE);
```

See Also:

Functions MCDoAction, MCSetActionFilter

MCDoAction mcActionSetKeysEnabled

## *MCDoAction* mcActionGetLooping

Syntax

```
ComponentResult MCDoAction(MovieController mcController, UINT
uAction, LPVOID lpvParams)
```

MCDoAction with the `mcActionGetLooping` determines whether looping is enabled for a movie controller

Parameters

*MovieController* mcController

The movie controller object.

*UINT* uAction

mcActionGetLooping

*LPVOID* lpvParams

A pointer to a boolean, set to TRUE if looping is enabled, FALSE if not.

Return

Zero if no error condition. Non-zero if error condition. See Appendix A for error condition values.

Comments

A movie controller with looping enabled plays a movie continuously, starting over at the beginning of the movie when the end is reached. Palindrome looping makes the movie play backward to the beginning before starting over.

Example

```
MovieController mcController;
BOOL bLoop;
•
•
// Turn looping on for a movie if it is off

   MCDoAction (mcController, mcActionGetLooping,
      (LPVOID) &bLoop);
   if (!bLoop)
      MCDoAction (mcController, mcActionSetLooping,
         (LPVOID) TRUE);
```

See Also:

Functions

MCDoAction, MCSetActionFilter

MCDoAction

mcActionSetLooping, mcActionSetLoopIsPalindrome

## *MCDoAction* mcActionGetLoopIsPalindrome

Syntax

ComponentResult MCDoAction (MovieController mcController, UINT uAction, LPVOID lpvParams)

MCDoAction with the mcActionGetLoopIsPalindrome determines whether palindrome looping is enabled for a movie controller.

Parameters

*MovieController* mcController

The movie controller object.

*UINT* uAction

mcActionGetLoopIsPalindrome

*LPVOID* lpvParams

A pointer to a boolean, set to TRUE if palindrome looping is enabled, FALSE if not.

Return — Zero if no error condition. Non-zero if error condition. See Appendix A for error condition values.

Comments — A movie controller with looping enabled plays a movie continuously, starting over at the beginning of the movie when the end is reached. Palindrome looping makes the movie play backward to the beginning when it reaches the end. Normal looping must also be enabled in order for palindrome looping to work.

Example

```
MovieController mcController;
BOOL bLoop;
•
•
// Turn palindrome looping on for a movie if it is off

   MCDoAction (mcController, mcActionGetLoopIsPalindrome,
      (LPVOID) &bLoop);
   if (!bLoop)
      {
      MCDoAction (mcController, mcActionSetLooping,
         (LPVOID) TRUE);
      MCDoAction (mcController, mcActionSetLoopIsPalindrome,
         (LPVOID) TRUE);
      }
```

See Also:

Functions — MCDoAction, MCSetActionFilter

MCDoAction — mcActionGetLooping, mcActionSetLooping, mcActionSetLoopIsPalindrome

## *MCDoAction* mcActionGetPlayEveryFrame

Syntax — ComponentResult MCDoAction (MovieController mcController, UINT uAction, LPVOID lpvParams)

MCDoAction with the mcActionGetPlayEveryFrame parameter determines if the movie controller has been instructed to play every frame in the movie.

Parameters — *MovieController* mcController

The movie controller object.

*UINT* uAction

mcActionGetPlayEveryFrame

*LPVOID* lpvParams

A pointer to a boolean, set to TRUE if movie controller set to play every frame in the movie, FALSE if not.

Return — Zero if no error condition. Non-zero if error condition. See Appendix A for error condition values.

Comments — If the movie is playing every frame, the sound will automatically be muted.

Example

```
MovieController mcController;
BOOL bPlay;
•
•
// See if every frame is being played. If not, make it so.

   MCDoAction (mcController, mcActionGetPlayEveryFrame,
      (LPVOID) &bPlay);
   if (!bPlay)
      MCDoAction
 (mcController, mcActionSetPlayEveryFrame,
        (LPVOID) TRUE);
```

See Also:

Functions — MCDoAction, MCSetActionFilter

MCDoAction — mcActionSetPlayEveryFrame

## *MCDoAction* mcActionGetPlayRate

Syntax — `ComponentResult MCDoAction (MovieController mcController, UINT uAction, LPVOID lpvParams)`

MCDoAction with the `mcActionGetPlayRate` parameter determines the movie's play rate.

Parameters

*MovieController* mcController

The movie controller object.

*UINT* uAction

mcActionGetPlayRate

*LPVOID* lpvParams

Pointer to an LFIXED play rate value.

Return — Zero if no error condition. Non-zero if error condition. See Appendix A for error condition values.

Comments

Rate of play values greater than 0 correspond to forward rates; values less than 0 play the movie backward. A value of 0 stops the movie. The integer portion of the LFIXED value is signed. The fractional part is not.

The LFIXED value is the rate of the movie expressed as a multiplier of the recorded rate. For example, a value of 1.0 means play the movie at the recorded rate. A value of 1.5 would mean play the movie one and 1/2 times faster than its recorded rate.

Use MCDoAction with mcActionPlay to set a movie's playback rate.

Example

```
MovieController mcController;
LFIXED lfxRate;
•
•
// Get the movie's play rate.

   McDoAction (mcController, mcActionGetPlayRate,
      (LPVOID) &lfxRate);
```

See Also:

Functions GetMoviePreferredRate, MCDoAction, MCSetActionFilter

MCDoAction mcActionPlay

## *MCDoAction* mcActionGetPlaySelection

Syntax

```
ComponentResult MCDoAction (MovieController mcController, UINT
uAction, LPVOID lpvParams)
```

MCDoAction with the `mcActionGetPlaySelection` parameter determines whether a movie is constrained to playing a selected portion of a movie.

Parameters

*`MovieController`* `mcController`

The movie controller object.

*`UINT`* `uAction`

`mcActionGetPlaySelection`

*`LPVOID`* `lpvParams`

A pointer to a boolean, set to `TRUE` if the movie will play only its selected portion, `FALSE` if not.

Return

Zero if no error condition. Non-zero if error condition. See Appendix A for error condition values.

Comments

A selection can be made and cleared using the movie controller. A darkened section of its slider represents the selected part of the movie.

Example

```
MovieController mcController;
BOOL bPlaySel;
•
•
// Turn off play selection if it is on

   MCDoAction (mcController, mcActionGetPlaySelection,
      (LPVOID) &bPlaySel);
   if (bPlaySel)
      MCDoAction (mcController, mcActionSetPlaySelection,
         (LPVOID) FALSE);
```

See Also:

Functions MCDoAction, MCSetActionFilter

MCDoAction mcActionSetPlaySelection, mcActionSetSelectionBegin, mcActionSetSelectionDuration

## *MCDoAction* mcActionGetUseBadge

Syntax

```
ComponentResult MCDoAction (MovieController mcController, UINT
uAction, LPVOID lpvParams)
```

MCDoAction with the `mcActionGetUseBadge` parameter determines whether a movie controller's ability to display a badge is enabled or disabled.

Parameters

*MovieController* mcController

The movie controller object.

*UINT* uAction

mcActionGetUseBadge

*LPVOID* lpvParams

A pointer to a boolean, set to `TRUE` if the badge can be used, `FALSE` if not.

Return

Zero if no error condition. Non-zero if error condition. See Appendix A for error condition values.

Comments

If a controller's badge capability is enabled, then the badge is displayed whenever the controller is not visible. When the controller is visible, the badge is not displayed. If the badge capability is disabled, the badge is never displayed.

Example

```
MovieController mcController;
BOOL bBadge;
•
•
// Turn on the badge if it is off

   MCDoAction (mcController, mcActionGetUseBadge,
```

```
    (LPVOID) &bBadge);
if (!bBadge)
   MCDoAction (mcController, mcActionSetUseBadge,
      (LPVOID) TRUE);
```

See Also:

Functions MCDoAction, MCSetActionFilter

MCDoAction mcActionSetUseBadge, mcActionBadgeClick

## *MCDoAction* mcActionGetVolume

Syntax `ComponentResult MCDoAction (MovieController mcController, UINT uAction, LPVOID lpvParams)`

MCDoAction with the `mcActionGetVolume` parameter retrieves the movie's volume.

Parameters *`MovieController`* `mcController`

The movie controller object.

*`UINT`* `uAction`

`mcActionGetVolume`

*`LPVOID`* `lpvParams`

A pointer to an SFIXED which will receive the volume.

Return Zero if no error condition. Non-zero if error condition. See Appendix A for error condition values.

Comments Volume ranges in value from -256 to +256. A negative value indicates the sound is muted, while preserving the absolute value of the volume.

Example

```
MovieController mcController;
SFIXED  sfxVolume;
•
•
// Get the movie's volume

   MCDoAction (mcController, mcActionGetVolume,
      (LPVOID) &sfxVolume);
```

See Also:

Functions GetMoviePreferredVolume, MCDoAction, MCSetActionFilter

MCDoAction mcActionSetVolume

## *MCDoAction* mcActionGoToTime

Syntax

```
ComponentResult MCDoAction(MovieController mcController, UINT
uAction, LPVOID lpvParams)
```

MCDoAction with the `mcActionGoToTime` parameter causes the movie to be positioned at the specified time value.

Parameters

*MovieController* mcController

The movie controller object.

*UINT* uAction

mcActionGoToTime

*LPVOID* lpvParams

The address of a time record specifying the position at which the movie will be set.

Return

Zero if no error condition. Non-zero if error condition. See Appendix A for error condition values.

Comments

The minimum `TimeValue` you can supply in the TimeRecord `poi`nted to in the third parameter which is the very beginning of the movie. The `TimeValue` is expressed in time units which are related to the movie's time scale.

The time coordinate system contains a time scale scored in time units. The number of units that pass per second quantifies the scale: a time scale of 26 means that 26 units pass per second and each time unit is 1/26 of a second.

When the duration of all or part of a movie is needed, it is expressed as a number of time units. Particular points in a movie can be identified by a time value, which is the number of time units to that point from the beginning of the movie.

Different movies may have different time scales. Use ConvertTimeScale to compare `TimeValues` between different movies.

Example

```
MovieController mcController;
TimeValue tvLocation;
Movie mMovie;
TimeRecord trRecord;
•
•
// Advance the movie to the poster frame

tvLocation = GetMoviePosterTime (mMovie);
trRecord.value.dwLo = tvLocation;
trRecord.value.dwHi = 0;
trRecord.scale = GetMovieTimeScale (mMovie);
MCDoAction(mcController, mcActionGoToTime,
   (LPVOID) &trRecord);
```

See Also:

Functions ConvertTimeScale, GetMoviePosterTime, GetMovieTimeScale, MCDoAction, MCGetCurrentTime, MCSetActionFilter

## *MCDoAction* mcActionIdle

Syntax

```
ComponentResult MCDoAction (MovieController mcController, UINT uAction, LPVOID lpvParams)
```

MCDoAction with the `mcActionIdle` parameter allocates processing time to a movie controller.

Parameters

*MovieController* mcController

The movie controller object.

*UINT* uAction

mcActionIdle

*LPVOID* lpvParams

NULL

Return Zero if no error condition. Non-zero if error condition. See Appendix A for error condition values.

Comments This action is used internally by QuickTime for Windows to keep movies playing. A filter you create can trap it and initiate further processing based on its being issued. In unusual cases where your program cannot use MCIsPlayerMessage, this action can be used directly to facilitate playing a movie.

Example See the sample program listing FILTERS.C in the QuickTime for Windows Tutorial section for further information about filters.

See Also:

Functions MCDoAction, MCIdle, MCSetActionFilter

## *MCDoAction* mcActionKey

Syntax

```
ComponentResult MCDoAction (MovieController mcController, UINT uAction, LPVOID lpvParams)
```

MCDoAction with the `mcActionKey` parameter causes a Windows `WM_KEYDOWN` or `WM_KEYUP` message to be passed to a movie controller.

Parameters

*MovieController* mcController

The movie controller object.

*UINT* uAction

mcActionKey

*LPVOID* lpvParams

The address of a Windows MSG structure..

Return — Zero if no error condition. Non-zero if error condition. See Appendix A for error condition values.

Comments — This action is normally issued by QuickTime for Windows internally when a key is pressed. A filter you create can trap it and initiate further processing based on its being issued. In unusual cases where your program cannot use MCIsPlayerMessage, this action could be used directly to facilitate playing a movie.

Example — See the sample program listing FILTERS.C in the QuickTime for Windows Tutorial section for further information about filters.

See Also:

Functions — MCDoAction, MCSetActionFilter, MCKey

## *MCDoAction mcActionPlay*

Syntax

```
ComponentResult MCDoAction (MovieController mcController, UINT
uAction, LPVOID lpvParams)
```

MCDoAction with the `mcActionPlay` parameter causes the movie to play at a specified play rate.

Parameters

*`MovieController`* `mcController`

The movie controller object.

*`UINT`* `uAction`

`mcActionPlay`

*`LPVOID`* `lpvParams`

LFIXED play rate value.

Return

Zero if no error condition. Non-zero if error condition. See Appendix A for error condition values.

Comments

Play rate values greater than 0 correspond to forward rates; values less than 0 play the movie backward. A value of 0 stops the movie. The integer portion of the LFIXED value is signed. The fractional part is not.

The LFIXED value is the rate of the movie expressed as a multiplier of the recorded rate. For example, a value of 1.0 means play the movie at its normal rate. A value of 1.5 would mean play the movie one and 1/2 times faster than its normal rate.

Use MCDoAction with mcActionGetPlayRate to determine a movie's playback rate.

Example

```
Movie mMovie;
MovieController mcController;
LFIXED lfxRate;
•
•
// Play the movie at 1.5 times its preferred rate.

   lfxRate = MAKELFIXED(0x0001, 0x8000);
   MCDoAction (mcController, mcActionPlay, (LPVOID) lfxRate);
```

See Also:

Functions

GetMoviePreferredRate, MCDoAction, MCSetActionFilter

MCDoAction

mcActionGetPlayRate

## *MCDoAction* mcActionSetFlags

Syntax

```
ComponentResult MCDoAction (MovieController mcController, UINT
uAction, LPVOID lpvParams)
```

MCDoAction with the `mcActionSetFlags` parameter sets a defined collection of flags that determine the behavior of the Movie Controller.

Parameters

*MovieController* mcController

The movie controller object.

*UINT* uAction

mcActionSetFlags

*LPVOID* lpvParams

A long integer that contains the flags to be set:

mcFlagsUseWindowPalette

mcFlagSuppressStepButtons

mcFlagSuppressSpeakerButton

Return

Zero if no error condition. Non-zero if error condition. See Appendix A for error condition values.

Comments

The following flags are defined:

mcFlagSuppressStepButtons - Determines whether the movie controller displays the step buttons. The step buttons allow the user to step the movie forward or backward one frame at a time. If this flag is set, the controller does not display the step buttons.

mcFlagSuppressSpeakerButton - Determines whether the movie controller displays the speaker button. The speaker button allows the user to control the movie's sound. If this flag is set, the controller does not display the speaker button.

mcFlagsUseWindowPalette - Determines whether the movie controller constructs a custom color palette, based on the color values found in the movie. This flag only works with display drivers that support palettes, typically those drivers that handle colors at pixel depth eight.

Example

```
MovieController mcController;
LONG lFlags;

// Show the speaker button

   MCDoAction (mcController, mcActionGetFlags, &lFlags);
   lFlags &= ~mcFlagSuppressSpeakerButton;
   MCDoAction (mcController, mcActionSetFlags, (LPVOID)
      lFlags);
```

See Also:

Functions

MCDoAction, MCSetActionFilter

MCDoAction

mcActionGetFlags

## *MCDoAction* mcActionSetGrowBoxBounds

Syntax

```
ComponentResult MCDoAction (MovieController mcController, UINT
uAction, LPVOID lpvParams)
```

MCDoAction with the `mcActionSetGrowBoxBounds` sets the size of the rectangle in which a movie can be resized.

Parameters

*MovieController* mcController

The movie controller object.

*UINT* uAction

mcActionSetGrowBoxBounds

*LPVOID* lpvParams

A pointer to the bounds rectangle which defines the new limits.

Return

Zero if no error condition. Non-zero if error condition. See Appendix A for error condition values.

Comments

Using an empty rectangle results in a movie controller not having a grow box. Using the current bounds rectangle (see MCGetControllerBoundsRect) allows resizing the movie smaller only. Using the client window rectangle allows resizing the movie up to the size of the client window.

Example

```
MovieController mcController;
RECT rcBounds;
•
•
// Allow resizing only less than current bounds

   MCGetControllerBoundsRect (mcController, &rcBounds);
   MCDoAction (mcController, mcActionSetGrowBoxBounds,
      (LPVOID) &rcBounds);
```

See Also:

Functions

MCDoAction, MCGetControllerBoundsRect, MCSetActionFilter

## *MCDoAction* mcActionSetKeysEnabled

Syntax

```
ComponentResult MCDoAction (MovieController mcController, UINT
uAction, LPVOID lpvParams)
```

MCDoAction with the `mcActionSetKeysEnabled` sets a movie controller's keyboard interface to the active or inactive state.

Parameters

*MovieController* mcController

The movie controller object.

*UINT* uAction

mcActionSetKeysEnabled

*LPVOID* lpvParams

A boolean, set to TRUE to enable a keyboard interface, FALSE to disable it.

Return — Zero if no error condition. Non-zero if error condition. See Appendix A for error condition values.

Comments — An inactive keyboard interface is the default attribute for a new movie controller. The keyboard interface will remain in an active state even if the controller is set inactive (i.e. grayed and unable to receive mouse clicks).

Example

```
MovieController mcController;
•
•
// Enable a movie controller's keyboard interface

   MCDoAction (mcController, mcActionSetKeysEnabled,
      (LPVOID) TRUE);
```

See Also:

Functions — MCDoAction, MCKey, MCSetActionFilter

MCDoAction — mcActionGetKeysEnabled, mcActionKey

## *MCDoAction* mcActionSetLooping

Syntax

```
ComponentResult MCDoAction (MovieController mcController, UINT
uAction, LPVOID lpvParams)
```

MCDoAction with the mcActionSetLooping parameter enables or disables looping for a movie controller.

Parameters

*MovieController* mcController

The movie controller object.

*UINT* uAction

mcActionSetLooping

*LPVOID* lpvParams

A boolean, set to TRUE to enable looping, FALSE to disable it.

Return — Zero if no error condition. Non-zero if error condition. See Appendix A for error condition values.

Comments — A movie controller with looping enabled plays a movie continuously, starting over at the beginning of the movie when the end is reached. Palindrome looping makes the movie play backward to the beginning before starting over.

Example

```
MovieController mcController;
•
•
// Turn looping on for a movie

   MCDoAction (mcController, mcActionSetLooping,
      (LPVOID) TRUE);
```

See Also:

Functions — MCDoAction, MCSetActionFilter

MCDoAction — mcActionGetLooping, mcActionSetLoopIsPalindrome

## *MCDoAction* mcActionSetLoopIsPalindrome

Syntax

```
ComponentResult MCDoAction (MovieController mcController, UINT
uAction, LPVOID lpvParams)
```

MCDoAction with the `mcActionSetLoopIsPalindrome` parameter enables or disables palindrome looping for a movie controller.

Parameters

*MovieController* mcController

The movie controller object.

*UINT* uAction

mcActionSetLoopIsPalindrome

*LPVOID* lpvParams

A boolean, set to `TRUE` to enable palindrome looping, `FALSE` to disable it.

Return — Zero if no error condition. Non-zero if error condition. See Appendix A for error condition values.

Comments — A movie controller with looping enabled plays a movie continuously, starting over at the beginning of the movie when the end is reached. Palindrome looping makes the movie play backward to the beginning when it reaches the end. Normal looping must also be enabled in order for palindrome looping to work.

Example

```
MovieController mcController;
•
```

```
•
// Turn palindrome looping on for a movie

   MCDoAction (mcController, mcActionSetLooping,
      (LPVOID) TRUE);
   MCDoAction (mcController, mcActionSetLoopIsPalindrome,
      (LPVOID) TRUE);
```

See Also:

Functions MCDoAction, MCSetActionFilter

MCDoAction mcActionSetLooping, mcActionGetLoopIsPalindrome

## *MCDoAction* mcActionSetPlayEveryFrame

Syntax `ComponentResult MCDoAction (MovieController mcController, UINT uAction, LPVOID lpvParams)`

MCDoAction with the `mcActionSetPlayEveryFrame` parameter instructs the movie controller to play every frame in the movie.

Parameters *`MovieController`* `mcController`

The movie controller object.

*`UINT`* `uAction`

`mcActionSetPlayEveryFrame`

*`LPVOID`* `lpvParams`

A boolean, set to `TRUE` to play every frame in the movie, `FALSE` to play movie frames normally.

Return Zero if no error condition. Non-zero if error condition. See Appendix A for error condition values.

Comments Issuing this instruction will automatically mute the movie's sound.

Example

```
MovieController mcController;
•
•
// Instruct the movie controller to play every frame

   MCDoAction (mcController, mcActionSetPlayEveryFrame,
      (LPVOID) TRUE);
```

See Also:

Functions MCDoAction, MCSetActionFilter

MCDoAction mcActionGetPlayEveryFrame, mcActionPlay

## *MCDoAction* mcActionSetPlaySelection

Syntax

```
ComponentResult MCDoAction (MovieController mcController, UINT
uAction, LPVOID lpvParams)
```

MCDoAction with the `mcActionSetPlaySelection` parameter constrains or unconstrains a movie controller to playing only the current selection.

Parameters

*MovieController* `mcController`

The movie controller object.

*UINT* `uAction`

`mcActionSetPlaySelection`

*LPVOID* `lpvParams`

A boolean, set to `TRUE` to constrain the controller to playing only its current selection, `FALSE` to unconstrain the controller.

Return

Zero if no error condition. Non-zero if error condition. See Appendix A for error condition values.

Comments

A selection can be made and cleared using the movie controller. A darkened section of its slider represents the selected part of the movie.

Example

```
MovieController mcController;
•
•
// Constrain playing to the selection

   MCDoAction (mcController, mcActionSetPlaySelection,
      (LPVOID) TRUE);
```

See Also:

Functions MCDoAction, MCSetActionFilter

MCDoAction mcActionGetPlaySelection, mcActionSetSelectionBegin, mcActionSetSelectionDuration

## *MCDoAction* mcActionSetSelectionBegin

Syntax

```
ComponentResult MCDoAction (MovieController mcController, UINT
uAction, LPVOID lpvParams)
```

MCDoAction with the `mcActionSetSelectionBegin` parameter sets the starting point of a selected portion of a movie.

Parameters

*MovieController* mcController

The movie controller object.

*UINT* uAction

mcActionSetSelectionBegin

*LPVOID* lpvParams

A pointer to a time record. You must specify the start time for the selection in the `TimeValue` field.

Return

Zero if no error condition. Non-zero if error condition. See Appendix A for error condition values.

Comments

This action has no effect unless a mcActionSetPlaySelection has been effected.

A selection can be made and cleared using the movie controller. A darkened section of its slider represents the selected part of the movie.

Example

```
MovieController mcController;
TimeRecord trRecord;
Movie mMovie;
TimeValue tvStart, tvDuration;
•
•
// Set the selection start time

   trRecord.value.dwLo = tvStart;
   trRecord.value.dwHi = 0;
   trRecord.scale = GetMovieTimeScale (mMovie);
   MCDoAction (mcController, mcActionSetSelectionBegin,
      (LPVOID) &trRecord);

// Set the selection duration

   trRecord.value.dwLo = tvDuration;
   trRecord.value.dwHi = 0;
   trRecord.scale = GetMovieTimeScale (mMovie);
   MCDoAction (mcController, mcActionSetSelectionDuration,
      (LPVOID) &trRecord);
```

See Also:

Functions

GetMovieActiveSegment, MCDoAction, MCSetActionFilter

mcActionSetSelectionDuration, mcActionSetPlaySelection

Data Types `TimeScale, TimeValue`

## *MCDoAction* mcActionSetSelectionDuration

Syntax

```
ComponentResult MCDoAction (MovieController mcController, UINT
uAction, LPVOID lpvParams)
```

MCDoAction with the `mcActionSetSelectionDuration` parameter sets the duration of a selected portion of a movie.

Parameters

*`MovieController`* `mcController`

The movie controller object.

*`UINT`* `uAction`

`mcActionSetSelectionDuration`

*`LPVOID`* `lpvParams`

The address of a time record. You must specify the duration of the selection in the `TimeValue` field.

Return

Zero if no error condition. Non-zero if error condition. See Appendix A for error condition values.

Comments

This action has no effect unless a mcActionSetPlaySelection has been effected. A selection can be made and cleared using the movie controller. A darkened section of its slider represents the selected part of the movie.

Example

```
MovieController mcController;
TimeRecord trRecord;
Movie mMovie;
TimeValue tvStart, tvDuration;
•
•
// Set the selection start time

   trRecord.value.dwLo = tvStart;
   trRecord.value.dwHi = 0;
   trRecord.scale = GetMovieTimeScale (mMovie);
   MCDoAction (mcController, mcActionSetSelectionBegin,
      (LPVOID) &trRecord);

// Set the selection duration

   trRecord.value.dwLo = tvDuration;
   trRecord.value.dwHi = 0;
   trRecord.scale = GetMovieTimeScale (mMovie);
   MCDoAction (mcController, mcActionSetSelectionDuration,
      (LPVOID) &trRecord);
```

See Also:

Functions GetMovieActiveSegment, MCDoAction, MCSetActionFilter

MCDoAction mcActionSetSelectionBegin, mcActionSetPlaySelection, mcActionSetSelectionDuration

Data Types TimeScale, TimeValue

## *MCDoAction* mcActionSetUseBadge

Syntax

```
ComponentResult MCDoAction (MovieController mcController, UINT
uAction, LPVOID lpvParams)
```

MCDoAction with the mcActionSetUseBadge parameter enables or disables a movie controller's ability to display a badge.

Parameters

*MovieController* mcController

The movie controller object.

*UINT* uAction

mcActionSetUseBadge

*LPVOID* lpvParams

A boolean, set to TRUE to enable the ability to display a badge, FALSE to disable it.

Return Zero if no error condition. Non-zero if error condition. See Appendix A for error condition values.

Comments If a controller's badge capability is enabled, then the badge is displayed whenever the controller is not visible. When the controller is visible, the badge is not displayed. If the badge capability is disabled, the badge is never displayed.

Example

```
MovieController mcController;
•
•
// Turn on the ability to display a badge

   MCDoAction (mcController, mcActionSetUseBadge,
      (LPVOID) TRUE);
```

See Also:

Functions MCDoAction, MCSetActionFilter

MCDoAction mcActionGetUseBadge

## *MCDoAction* mcActionSetVolume

Syntax

```
ComponentResult MCDoAction (MovieController mcController, UINT
uAction, LPVOID lpvParams)
```

MCDoAction with the `mcActionSetVolume` parameter sets the movie's volume.

Parameters

*MovieController* mcController

The movie controller object.

*UINT* uAction

mcActionSetVolume

*LPVOID* lpvParams

A SFIXED value indicating the volume.

Return

Zero if no error condition. Non-zero if error condition. See Appendix A for error condition values.

Comments

Volume ranges in value from -256 to +256. A negative value indicates the sound is muted, while preserving the absolute value of the volume.

Example

```
MovieController mcController;
Movie mMovie;
SFIXED sfxVolume;
•
•
// Set the movie's volume to its preferred level

   sfxVolume = GetMoviePreferredVolume (mMovie);
   MCDoAction (mcController, mcActionSetVolume,
      (LPVOID) sfxVolume);
```

See Also:

Functions

GetMoviePreferredVolume, MCDoAction, MCSetActionFilter

MCDoAction

mcActionGetVolume

## *MCDoAction* mcActionStep

Syntax

```
ComponentResult MCDoAction (MovieController mcController, UINT
uAction, LPVOID lpvParams)
```

MCDoAction with the `mcActionStep` parameter causes the movie to play a specified number of frames at a time.

Parameters

*MovieController* mcController

The movie controller object.

*UINT* uAction

mcActionStep

*LPVOID* lpvParams

A SHORT indicating the number of frames in the step.

Return — Zero if no error condition. Non-zero if error condition. See Appendix A for error condition values.

Comments — Using a positive number of frames steps the movie forward. Using a negative number steps the movie backward.

Example

```
MovieController mcController;
•
•
// Step the movie forward three frames

   MCDoAction (mcController, mcActionStep, (LPVOID) 3);
```

See Also:

Functions — MCDoAction, MCSetActionFilter

MCDoAction — mcActionPlay

### *MCDraw*

Syntax

```
ComponentResult MCDraw (MovieController mcController, HWND
hWnd)
```

MCDraw redraws the movie image.

Parameters

*MovieController* mcController

The movie controller object.

*HWND* hWnd

The handle to the window.

Return — Zero if no error condition. Non-zero if error condition. See Appendix A for error condition values.

Comments — MCDraw calls MCDoAction with mcActionDraw. MCDraw is typically used to manually refresh the movie image.

Example

```
MovieController mcController;
HWND hWnd;
```

```
•
•
MCDraw (mcController, hWnd);
```

See Also:

Functions MCIsPlayerMessage

MCDoAction mcActionDraw

## *MCDrawBadge*

Syntax

```
ComponentResult MCDrawBadge (MovieController mcController, HRGN
hrgnMovieRgn, HRGN FAR *lphrgnBadgeRgn)
```

`MCDrawBadge` displays a movie controller's badge.

Parameters

*`MovieController`* `mcController`

The movie controller object.

*`HRGN`* `hrgnMovieRgn`

Must be set to NULL

*`HRGN FAR`* `*lphrgnBadgeRgn`

The address of the handle to a windows region which will be set to the region occupied by the badge. If called as NULL, no region is returned.

Return

Zero if no error condition. Non-zero if error condition. See Appendix A for error condition values. The window region referenced by the third parameter will be populated with information about the badge.

Comments

The second parameter must be NULL.

A badge may be drawn whether the movie is paused or playing. Any new movie frame, however, will overlay it. The recommended method for displaying a badge is to specify the `mcWithBadge` flag when NewMovieController is called, which will display it automatically when the movie controller is hidden. You can also enable a controller to display a badge by using MCDoAction with mcActionSetUseBadge.

`MCDrawBadge` ignores the `mcWithBadge` flag and will work even if the flag was not specified when the movie controller was created.

MCSetVisible also may be used to draw a badge by side effect, if the movie controller's visibility is set to `FALSE` and its badge flag is turned on.

Example

```
MovieController mcController;
HRGN hrgnBadge;
•
•
MCDrawBadge (mcController, NULL, &hrgnBadge);
```

See Also:

Functions MCSetVisible

MCDoAction mcActionGetUseBadge, mcActionSetUseBadge

## *MCGetControllerBoundsRect*

Syntax

```
ComponentResult MCGetControllerBoundsRect (MovieController
mcController, LPRECT lprcBounds)
```

`MCGetControllerBoundsRect` retrieves the bounds rectangle of the movie and movie controller, or just the controller, depending on whether they are attached or detached.

Parameters

*`MovieController`* `mcController`

The movie controller object.

*`LPRECT`* `lprcBounds`

The address of the bounds rectangle.

Return

Zero if no error condition. Non-zero if error condition. See Appendix A for error condition values. The bounds rectangle is populated with the bounds coordinates.

Comments

If the movie controller is attached to the movie, the bounds rectangle referenced by the second parameter is the smallest rectangle completely encompassing both the movie and movie controller. When a controller is detached, its dimensions alone determine the bounds rectangle.

*Note: All QuickTime for Windows routines referencing a RECT or POINT assume client device coordinates.*

Example

```
RECT rcBounds;
MovieController mcController;
•
•
MCGetControllerBoundsRect (mcController, &rcBounds);
```

See Also:

Functions MCNewAttachedController, MCSetControllerAttached, MCSetControllerBoundsRect

## *MCGetControllerInfo*

Syntax

```
ComponentResult MCGetControllerInfo (MovieController
mcController, LPLONG lplMcInfoFlags)
```

`MCGetControllerInfo` determines the current status of a set of movie controller

flags.

Parameters

*MovieController* mcController

The movie controller object.

*LPLONG* lplMCInfoFlags

The address of a long integer which will contain the bit flags denoting various movie controller attributes:

mcInfoHasSound

mcInfoIsPlaying

mcInfoIsLooping

mcInfoIsInPalindrome

Return

Zero if no error condition. Non-zero if error condition. See Appendix A for error condition values. The long integer referenced by the second parameter will be populated with flag settings indicating controller attributes.

Comments

The MCInfoFlags are defined as follows:

mcInfoHasSound – Indicates the movie has a sound track.

mcInfoIsPlaying – Indicates the movie was playing when the call was made.

mcInfoIsLooping – Indicates the controller was playing the movie in looping mode when the call was made.

mcInfoIsInPalindrome – Indicates the controller was playing the movie in palindrome mode when the call was made.

Example

```
MovieController mcController;
LONG lMCInfoFlags;
•
•
// See if the movie has sound

   MCGetControllerInfo (mcController, &lMCInfoFlags);
   if (lMCInfoFlags & mcInfoHasSound)
      /* Appropriate action if movie has sound. */
   else
      /* Appropriate action if movie has no sound. */
```

See Also:

Functions MCDoAction

MCDoAction mcActionSetLooping, mcActionSetLoopIsPalindrome, mcActionPlay

## *MCGetCurrentTime*

Syntax

```
TimeValue MCGetCurrentTime (MovieController mcController,
TimeScale FAR *tsScale)
```

`MCGetCurrentTime` retrieves the time value represented by the slider control on the movie controller. It can also be used to obtain the time scale for this time value.

Parameters

*`MovieController`* `mcController`

The movie controller object.

*`TimeScale FAR`* `*tsScale`

A pointer to the `TimeScale` value. May be set to NULL if it is not needed.

Return

The `TimeValue` represented by the slider on the controller. If there are no movies associated with the controller, the returned `TimeValue` is set to zero.

Comments

This function may be called whether a movie is playing or not.

Example

```
Movie mMovie;
MovieController mcController;
PicHandle phPicture;
RECT rcPicture;
HDC hdc;
TimeValue tvTime;
•
•
// Retrieve frame at current movie time plus two seconds

   tvTime = MCGetCurrentTime (mcController, NULL) +
      (2 * GetMovieTimeScale (mMovie));
   if ((phPicture = GetMoviePict (mMovie, tvTime)) != NULL)
      DrawPicture (hdc, phPicture, &rcPicture, NULL);
```

See Also:

Functions

MCDoAction

MCDoAction

mcActionGoToTime

Data Types

`TimeScale, TimeValue`

## *MCGetMovie*

Syntax

```
Movie MCGetMovie (MovieController mcController)
```

`MCGetMovie` retrieves the movie object associated with a specified movie controller.

Parameters

*`MovieController`* `mcController`

The movie controller object.

Return

The movie object associated with the movie controller. NULL is returned if no movie is

associated with the controller.

Comments The associated movie object is retrieved whether the controller is attached or not.

Example

```
MovieController mcController;
Movie mMovie;
•
•
mMovie = MCGetMovie (mcController);
```

See Also:

Functions MCSetMovie

### *MCGetVisible*

Syntax `ComponentResult MCGetVisible (MovieController mcController)`

`MCGetVisible` determines whether a movie controller is visible.

Parameters *`MovieController`* `mcController`

The movie controller object.

Return `FALSE` if the movie controller is invisible. `TRUE` if the movie controller is visible. See Appendix A for error condition values.

Comments Use the function MCSetVisible to make a movie controller visible or invisible.

Example

```
MovieController mcController;
•
•
// Make controller invisible if it is visible

   if (MCGetVisible (mcController))
      {
      MCSetVisible (mcController, FALSE);
      }
```

See Also:

Functions MCSetVisible, MCActivate

### *MCIdle*

Syntax `ComponentResult MCIdle (MovieController mcController)`

`MCIdle` is used to keep a movie playing when your program is unable to use MCIsPlayerMessage.

Parameters *`MovieController`* `mcController`

The movie controller object.

Return — Zero if no error condition. Non-zero if error condition. See Appendix A for error condition values.

Comments — `MCIdle` calls MCDoAction with mcActionIdle. Using the routine MCIsPlayerMessage is the recommended method to keep a movie playing, and you should use `MCIdle` only in special circumstances where you must micro-manage the movie controller or cannot use MCIsPlayerMessage.

See Also:

Functions — MCDoAction, MCIsPlayerMessage

MCDoAction — mcActionIdle

## *MCIsControllerAttached*

Syntax

```
ComponentResult MCIsControllerAttached  (MovieController
mcController)
```

`MCIsControllerAttached` determines whether a movie controller is attached to a movie.

Parameters — *`MovieController`* `mcController`

The movie controller object.

Return — `TRUE` if the controller is attached, `FALSE` if not. Otherwise an error condition. See Appendix A for error condition values.

Comments — Use the MCSetControllerAttached function to attach or detach a movie controller. Remember not to confuse attachment with association. An attached controller is physically adjacent to the movie on the screen. An associated controller is used to run a movie, and need not be attached.

Example

```
MovieController mcController;
RECT rcMovie, rcController;
•
•
// Detach the controller and move it away from movie
// But only if it is attached

   if (MCIsControllerAttached (mcController))
      {
      MCSetControllerAttached (mcController, FALSE);
      MCPositionController (mcController, &rcMovie,
         &rcController, 0L);
      }
```

See Also:

Functions MCPositionController, MCSetControllerAttached

## *MCIsPlayerMessage*

Syntax

```
ComponentResult MCIsPlayerMessage (MovieController mcController,
HWND hWnd, UINT wMessage, WPARAM wParam, LPARAM lParam)
```

`MCIsPlayerMessage` is the routine normally used to keep a movie playing. It is called in a program's window procedure and redirects all messages targeted for the movie controller.

Parameters

*`MovieController`* `mcController`

The movie controller object.

*`HWND`* `hWnd`

The argument received by the window procedure.

*`UINT`* `wMessage`

The argument received by the window procedure.

*`WPARAM`* `wParam`

The argument received by the window procedure.

*`LPARAM`* `lParam`

The argument received by the window procedure.

Return

If a message received by the window procedure is not meant for the movie controller, `MCIsPlayerMessage` returns `FALSE` and the message gets processed normally. If the message is handled by the movie controller, `MCIsPlayerMessage` returns `TRUE`.

Comments

For each movie controller you create, you will have to code a separate call to `MCIsPlayerMessage` with the corresponding movie controller object as the first parameter

`MCIsPlayerMessage` is not the only method of playing a movie. However, it is highly recommended. See the descriptions of MCIdle and MCKey.

Example

```
LONG FAR PASCAL WndProc (HWND hWnd, UINT msg, WPARAM wParam,
   LPARAM lParam)
   {
// Drive the movie controller

   if (MCIsPlayerMessage (mcController, hWnd, msg, wParam, lParam))
      return 0;

// Process the windows message

   switch (msg)
```

```
        {
        •
        •
        }
    }
```

See Also:

Functions MCIdle, MCKey

## *MCKey*

Syntax `ComponentResult MCKey (MovieController mcController, WPARAM wParam, LPARAM lParam);`

`MCKey` calls MCDoAction with mcActionKey, which causes a Windows `WM_KEYDOWN` or `WM_KEYUP` message to be passed to a movie controller.

Parameters *`MovieController`* `mcController`

The movie controller object.

*`WPARAM`* `wParam`

The argument received by the window procedure.

*`LPARAM`* `lParam`

The argument received by the window procedure.

Return Zero if no error condition. Non-zero if error condition. See Appendix A for error condition values.

Comments `MCKey` and MCIdle can be used instead of MCIsPlayerMessage when your program is unable to use that particular function.

See Also:

Functions MCIsPlayerMessage, MCIdle

## *MCNewAttachedController*

Syntax `ComponentResult MCNewAttachedController (MovieController mcController, Movie mMovie, HWND hWnd, POINT ptUpperLeft)`

`MCNewAttachedController` attaches an existing movie to an existing movie controller.

Parameters *`MovieController`* `mcController`

The existing movie controller object.

*Movie* mMovie

The existing movie object.

*HWND* hWnd

The parent window handle.

*POINT* ptUpperLeft

The upper left corner of the movie rectangle.

Return — Zero if no error condition. Non-zero if error condition. See Appendix A for error condition values.

Comments — When a movie is associated with a movie controller, a reference to the movie object is recorded in the controller's data structure. Movie data structures contain no elements which link them with movie controllers. The point specified by ptUpperLeft becomes the new upper left corner of the bounds rectangle.

Once a movie is associated with a controller, it starts playing immediately (assuming it has a non-zero play rate, which is normally the case). To make a movie paused when first visible and associated with a new controller, you can use MCDoAction with an action of mcActionPlay and a play rate of 0. It is good style to do this as soon as possible after performing the association.

*Note: All QuickTime for Windows routines referencing a RECT or POINT assume client device coordinates.*

Example

```
Movie mMovie;
MovieController mcController;
POINT ptUpperLeft;
•
•
MCNewAttachedController (mcController, mMovie, hWnd,
  ptUpperLeft);
```

See Also:

Functions — NewMovieController, MCSetMovie

## *MCPositionController*

Syntax

```
ComponentResult MCPositionController (MovieController
mcController, LPRECT lprcMovieRect, LPRECT lprcControllerRect,
LONG lControllerCreationFlags)
```

MCPositionController sets the size and position of a movie and its controller. This function works with both attached and detached movie controllers.

Parameters — *MovieController* mcController

The movie controller object.

*LPRECT* lprcMovieRect

The address of a RECT structure specifying the coordinates of the movie's bounds rectangle.

*LPRECT* lprcControllerRect

The address of a RECT structure specifying the coordinates of the controller's bounds rectangle. Use NULL if the movie controller is attached.

*LONG* lControllerCreationFlags

A LONG containing flags that modify the result of the routine. These are the same flags used with NewMovieController.

If you set this parameter to 0, the movie will be centered in the movie rectangle and the movie will be scaled to fit in that rectangle. These flags are:

mcTopLeftMovie – Places the movie at the top left hand corner of the movie rectangle specified.

mcScaleMovieToFit – Resizes the movie to fit into the movie rectangle specified (excluding the area taken up by the controller).

Return

Zero if no error condition. Non-zero if error condition. See Appendix A for error condition values.

Comments

This is the recommended call to reposition and resize a movie with a detached controller. Remember not to confuse attachment with association. An attached controller is physically adjacent to the movie.

An associated controller is used to run a movie, and need not be attached.

Whenever the controller bounds rectangle changes, your action filter, if any, will get called with mcActionControllerSizeChanged after the changes to the rectangle have occurred.

*Note: All QuickTime for Windows routines referencing a RECT or POINT assume client device coordinates.*

Example

```
MovieController mcController;
RECT rcMovie, rcController;
•
•
// Detach the controller and move it away from movie

   MCSetControllerAttached (mcController, FALSE);
   MCPositionController (mcController, &rcMovie,
      &rcController, 0L);
•
•
// Re-attach the controller

   MCSetControllerAttached (mcController, TRUE);
```

See Also:

Functions MCIsControllerAttached, MCSetControllerAttached, NewMovieController

## *MCSetActionFilter*

Syntax

```
ComponentResult MCSetActionFilter (MovieController mcController,
MCActionFilter lpfnFilter, LONG lRefCon)
```

`MCSetActionFilter` sets an action filter function for a movie controller.

Parameters

*`MovieController`* `mcController`

The movie controller object.

*`MCActionFilter`* `lpfnFilter`

The address of the user-defined filter function.

*`LONG`* `lRefCon`

Additional data of use to the filter when processing the action. Should be coded as `0L` if not used.

Return

Zero if no error condition. Non-zero if error condition. See Appendix A for error condition values.

Comments

An action filter intercepts the MCDoAction call, providing the opportunity to process the action item before the movie controller.

The filter function must return a boolean: `TRUE` indicates the controller doesn't have to handle the action. `FALSE` tells the controller to complete any appropriate processing of the action item.

To remove the filter, you must call `MCSetActionFilter` with the filter function address set to NULL.

If you compile your program using Borland *smart callbacks* or Microsoft's `-GEs` compiler option, or your filter function is in a dynamic link library, you do not need to use `MakeProcInstance` on your filter address before calling `MCSetActionFilter`.

Example

```
// Filter function declaration

   BOOL CALLBACK __export MyFilter (MovieController mcController,
      UINT uAction, LPVOID lpParam, LONG lRefCon);

-------------------------------------------------------------------

// The application window procedure

   MovieController mcController;
   struct {...} *pData;
   •
   •
   MCSetActionFilter (mcController, MyFilter, (LONG) pData);
```

```
// The filter function

   BOOL CALLBACK __export MyFilter (MovieController mcController,
      UINT uAction, LPVOID lpParam, LONG lRefCon)
      {
      PVOID pStruct;

      switch (uAction)
         {
         case mcActionControllerSizeChanged:

            pStruct = (PVOID) lRefCon;

         /* Do something with structure whose address was passed. */
            •
            •
            return TRUE;

         default:
            return FALSE;
         }
      }
```

See Also:

Functions MCDoAction, MCActionFilter

## *MCSetControllerAttached*

Syntax `ComponentResult MCSetControllerAttached (MovieController mcController, BOOL bAttach)`

`MCSetControllerAttached` attaches or detaches a movie controller from a movie.

Parameters *`MovieController`* `mcController`

The movie controller object.

*`BOOL`* `bAttach`

`TRUE` attaches the movie controller, `FALSE` detaches it.

Return Zero if no error condition. Non-zero if error condition. See Appendix A for error condition values.

Comments Remember not to confuse attachment with association. An attached controller is physically adjacent to the movie on the screen. An associated controller is used to run a movie, and need not be attached.

If the controller is physically removed from the movie prior to attachment, it will jump to its normal attached position directly below the movie when `MCSetControllerAttached` is executed with `TRUE`.

Example

```
MovieController mcController;
RECT rcMovie, rcController;
•
•
// Detach the controller and move it away from movie

   MCSetControllerAttached (mcController, FALSE);
   MCPositionController (mcController, &rcMovie,
      &rcController, 0L);

// Re-attach the controller to the movie

   MCSetControllerAttached (mcController, TRUE);
```

See Also:

Functions MCIsControllerAttached, MCPositionController

### *MCSetControllerBoundsRect*

Syntax

```
ComponentResult MCSetControllerBoundsRect (MovieController
mcController, const LPRECT lprcBounds)
```

`MCSetControllerBoundsRect` resets the dimensions of a movie controller. If the controller is attached, the movie may be resized as well.

Parameters

*`MovieController`* `mcController`

The movie controller object.

*`const LPRECT`* `lprcBounds`

The address of the new bounds rectangle.

Return

Zero if no error condition. Non-zero if error condition. See Appendix A for error condition values.

Comments

When a movie controller is detached, its dimensions alone will be determined by the new bounds rectangle. A movie controller's height cannot be reset. If the rectangle has a height larger than the standard controller height, the movie controller is centered vertically.
*Note: All QuickTime for Windows routines referencing a RECT or POINT assume client device coordinates.*

When a movie controller is attached, the controller will use part of the new bounds rectangle for itself. The movie will be sized to fit the remaining portion of the rectangle.

Whenever the controller bounds rectangle changes, your action filter, if any, will get called with mcActionControllerSizeChanged after the changes to the rectangle have occurred.

Example

```
RECT rcBounds;
MovieController mcController;
•
```

```
MCSetControllerBoundsRect (mcController, &rcBounds);
```

See Also:

Functions MCGetControllerBoundsRect, MCNewAttachedController, MCSetControllerAttached

## *MCSetMovie*

Syntax `ComponentResult MCSetMovie (MovieController mcController, Movie mMovie, HWND hWndMovieWindow, POINT ptUpperLeft)`

`MCSetMovie` associates or disassociates an existing movie controller with an existing movie. If the `mMovie` parameter is set to NULL, the movie controller is not associated with any movie.

Parameters *`MovieController`* `mcController`

The existing movie controller object.

*`Movie`* `mMovie`

The existing movie object.

*`HWND`* `hWndMovieWindow`

The parent window handle.

*`POINT`* `ptUpperLeft`

A new location on the screen for the movie controller bounds rectangle.

Return Zero if no error condition. Non-zero if error condition. See Appendix A for error condition values.

Comments `MCSetMovie` is identical to MCNewAttachedController except that it is possible to specify NULL as the movie object. The point specified by `ptUpperLeft` becomes the new upper left corner of the bounds rectangle. This routine is the best way to associate a different movie with a controller.

If appropriate, the location of the controller can be changed. When the movie controller is attached, this moves the movie to another location of the screen.

When a controller is associated with a movie, a reference to the movie object is recorded in the controller's data structure. A movie controller can be associated with many movies during its existence, but only one at a time. Movie data structures contain no elements which link them with movie controllers.

Movie controllers remain associated with movies regardless of their states. If a controller is made invisible or inactive, for instance, it stays associated with its movie. Conversely, movies continue to play even if the states of their associated controllers are changed while they are playing. If either one of an associated pair is destroyed,

the other is not affected.

Once a movie is associated with a controller, it starts playing immediately (assuming it has a non-zero play rate, which is normally the case). To make a movie paused when first visible and associated with a new controller, you can use MCDoAction with an action of mcActionPlay and a play rate of 0. It is good style to do this as soon as possible after performing the association.

Association implies nothing about the proximity of movies and their controllers on the screen. It is simply the means by which any movie can be plugged in to any controller and played.

Whenever the controller bounds rectangle changes, your action filter, if any, will be called with mcActionControllerSizeChanged after the changes to the rectangle have occurred.

*Note: All QuickTime for Windows routines referencing a RECT or POINT assume client device coordinates.*

Example

```
MovieController mcController;
POINT ptUpperLeft;
•
•
// Disassociate the movie controller from its movie

   MCSetMovie (mcController, NULL, hWnd, ptUpperLeft);
```

See Also:

Functions NewMovieController, MCNewAttachedController, MCSetControllerAttached

## *MCSetVisible*

Syntax

```
ComponentResult MCSetVisible (MovieController mcController,
BOOL bShow)
```

`MCSetVisible` hides a visible movie controller and makes visible a hidden movie controller.

Parameters

*`MovieController`* `mcController`

The movie controller object.

*`BOOL`* `bShow`

`TRUE` makes the movie controller visible, `FALSE` hides it.

Return

Zero if no error condition. Non-zero if error condition. See Appendix A for error condition values.

Comments

Invisible movie controllers can be attached, detached, active or inactive. You just can't see them. To query the visibility state of a movie controller, use MCGetVisible.

Calling `MCSetVisible` with `FALSE` displays the badge if the badge flag is turned on. See the description of MCDrawBadge for more information about badges.

Example

```
MovieController mcController;
•
•
// Hide the movie controller

   MCSetVisible (mcController, FALSE);
```

See Also:

Functions MCDrawBadge, MCGetVisible, NewMovieController

### *NewMovieController*

Syntax

```
MovieController NewMovieController (Movie mMovie, const LPRECT
lprcMovieRect, LONG lControllerCreationFlags, HWND hWndParent)
```

`NewMovieController` creates and attaches a movie controller to a movie.

Parameters

*Movie* mMovie

The movie object to be associated with the new movie controller. This movie object was assigned by QuickTime for Windows when it processed NewMovieFromFile. It can be NULL, which means that the new controller will not be associated with any movie.

*const LPRECT* lprcMovieRect

The address of a bounds rectangle which will determine the movie and movie controller's size and position, depending on the creation flags.

*LONG* lControllerCreationFlags

A `LONG` containing flags that modify the result of the routine. If you set this parameter to 0, the movie will be centered in the movie rectangle and the movie will be scaled to fit in that rectangle. These flags are:

`mcScaleMovieToFit` - Resizes the movie to fit into the movie rectangle specified (excluding the area taken up by the controller).

both `mcTopLeftMovie` and `mcScaleMovieToFit` - resizes the movie to fit into the movie rectangle specified, then expands the bounds rectangle to include the movie controller (without cutting into the movie area).

`mcWithBadge` - Determines whether the controller can display a badge.

`mcNotVisible` - Determines the initial visibility state of the movie controller.

`mcTopLeftMovie` - Places the movie at the top left hand corner of the movie rectangle specified.

*HWND* hWndParent

The parent window handle of the new movie controller.

Return

A `MovieController` object. NULL indicates an error condition.

Comments

`NewMovieController` creates the new controller within the bounds rectangle even when the movie object is NULL. For all but one configuration of the controller creation flags, the movie controller takes a portion out of the specified rectangle. The exception is when both `mcTopLeftMovie` and `mcScaleMoveToFit` are specified, in which case the movie controller is connected abutting the specified bounds rectangle.

To display the movie at optimum size with the correct aspect ratio, call GetMovieBox before `NewMovieController`, and use the retrieved rectangle as the bounds

rectangle. Then specify both the `mcTopLeftMovie` and `mcScaleMoveToFit` flags. Use the `mcWithBadge` flag to enable badge availability. This is the recommended method of working with badges.

Movies and movie controllers are not permanently associated. Movie controllers can be dynamically reassigned to movies at any point in the program provided they are properly initialized. Destroying one does not destroy the other, nor does disconnecting a movie from a movie controller disable either component.

When a controller is associated with a movie, a reference to the movie object is recorded in the controller's data structure. A movie controller can be associated with many movies during its existence, but only one at a time. Movie data structures contain no elements which link them with movie controllers.

Once a movie is associated with a controller, it starts playing immediately (assuming it has a non-zero play rate, which is normally the case). To make a movie paused when first visible and associated with a new controller, you can use MCDoAction with an action of mcActionPlay and a play rate of 0. It is good style to do this as soon as possible after performing the association.

To play *n* cases of the same movie simultaneously, the movie file must be opened *n* times to get *n* unique movie objects and then create or associate *n* movie controllers.

*Note: All QuickTime for Windows routines referencing a RECT or POINT assume client device coordinates.*

Example

```
Movie mMovie;
MovieController mcController;
HWND hWndParent;
RECT rcMovie;
•
•
// Instantiate movie controller
// Movie to display at optimum size & aspect ratio

   GetMovieBox (mMovie, &rcMovie);
   OffsetRect (&rcMovie, -rcMovie.left, -rcMovie.top);
   mcController = NewMovieController (mMovie, &rcMovie,
      mcTopLeftMovie + mcScaleMovieToFit, hWndParent);
```

See Also:

Functions DisposeMovieController, MCNewAttachedController, MCSetMovie

## *NewMovieFromDataFork*

Syntax

```
OSErr NewMovieFromDataFork (Movie FAR *fpmMovie, HFILE hFile,
LONG lOffset, UINT uiNewMovieFlags)
```

`NewMovieFromDataFork` initializes a movie object and associated storage in the same manner as NewMovieFromFile, except that movie data is retrieved from an open DOS file, beginning at a specified offset.

Parameters

*`Movie FAR *`*`fpmMovie`

The address of the movie object to be allocated.

*HFILE* hFile

The file handle of an open DOS file containing the movie data.

*LONG* lOffset

An offset into the DOS file representing the start of the movie data.

*UINT* uiNewMovieFlags

newMovieActive sets movie active, 0 sets it inactive.

Return

Zero if no error condition. Non-zero if error condition. See Appendix A for error condition values. You can use GetMoviesError and GetMoviesStickyError to test for failure of this call.

Comments

This routine provides an alternative to NewMovieFromFile when movie data is stored in a non-standard movie file. Note that the movie object will be in a non-active state when it is extracted.

Also be aware that, unlike NewMovieFromFile, you must not close the DOS file containing the movie until after you have called DisposeMovie.

Example

```
Movie mMovie;
OFSTRUCT ofstruct;
LONG lOffset
HFILE fhHandle;
   •
   •
// Open the DOS file containing the movie data

   fhHandle = OpenFile ("NEWSREEL.BIN", &ofstruct, OF_READ);

// Extract a movie object

   NewMovieFromDataFork (&mMovie, fhHandle, lOffset);
   •
   •
// Free the movie memory

   DisposeMovie (mMovie);

// Close the DOS file

   _lclose (fhHandle);
```

See Also:

Functions

OpenMovieFile, CloseMovieFile, GetMoviesError, GetMoviesStickyError, NewMovieFromFile

## *NewMovieFromFile*

Syntax

```
OSErr NewMovieFromFile (Movie FAR *fpmMovie, MovieFile mfMovie,
SHORT FAR *lpsResID, LPSTR lpstrResName, UINT uiNewMovieFlags,
BOOL FAR *lpbDataRefWasChanged)
```

`NewMovieFromFile` initializes a movie object, allocates and initializes all storage required for the movie and performs various internal tasks such as telling QuickTime for Windows' scheduler to add the movie to its tables.

Parameters

*`Movie FAR *`*`fpmMovie`

The address of the movie object.

*`MovieFile`* `mfMovie`

The reference value that refers to the open movie file. This is obtained from OpenMovieFile.

*`SHORT FAR *`*`lpsResID`

Set to NULL.

*`LPSTR`* `lpstrResName`

Set to NULL.

*`UINT`* `uiNewMovieFlags`

`newMovieActive` sets movie active, 0 sets it inactive.

*`BOOL FAR *`*`lpbDataRefWasChanged`

Set to NULL.

Return

Zero if no error condition. Non-zero if error condition. See Appendix A for error condition values. You can use GetMoviesError and GetMoviesStickyError to test for failure of this call.

Comments

For each movie you wish to play, you must call OpenMovieFile followed by `NewMovieFromFile`. As soon as possible after `NewMovieFromFile`, the movie file may be closed with CloseMovieFile.

To play *n* cases of the same movie simultaneously, the movie file must be opened *n* times to get *n* unique movie objects and then associated with *n* movie controllers.

Example

```
MovieFile mfMovie;
Movie mMovie;
MovieController mcController;
RECT rcMovie;
•
•
// Open the movie file

   OpenMovieFile ("NEWSREEL.MOV", &mfMovie, OF_READ);
```

```
// Establish a movie object

   NewMovieFromFile (&mMovie, mfMovie, NULL, NULL, 0, NULL);

// Close the movie file

   CloseMovieFile (mfMovie);

// Get a bounds rectangle

   GetMovieBox (mMovie, &rcMovie);
   OffsetRect (&rcMovie, -rcMovie.left, -rcMovie.top);

// Create a movie controller

   mcController = NewMovieController (mMovie, &rcMovie,
      mcTopLeftMovie + mcScaleMovieToFit, hWndParent);

// Make the movie active

   SetMovieActive (mMovie, TRUE);
```

See Also:

Functions OpenMovieFile, CloseMovieFile, GetMoviesError, GetMoviesStickyError

## *NormalizeRect*

Syntax `VOID NormalizeRect (LPRECT lprcRect)`

`NormalizeRect` adjusts the width and height of a rectangle such that its aspect ratio matches that of a similar rectangle on the Macintosh.

Parameters *`LPRECT`* `lprcRect`

The address of the rectangle to normalize.

Return None. The normalized rectangle is placed in the rectangle referenced. You can use GetMoviesError and GetMoviesStickyError to test for failure of this call.

Comments `NormalizeRect` uses the LOGPIXELSX and LOGPIXELSY values returned from the Windows function `GetDeviceCaps` to adjust the width and height of a rectangle. It ensures the correct aspect ratio of the movie rectangle.

*Note: All QuickTime for Windows routines referencing a RECT or POINT assume client device coordinates.*

Example

```
PicFile pfPicture;
OpenCPicParams ocppHeader;
OFSTRUCT ofsOpenFileStr;
RECT rcFrame;
•
•
```

```
OpenPictureFile ("HOUSE.PIC", &pfPicture, OF_READ);
GetPictureFileHeader  (pfPicture, &rcFrame, &ocppHeader);
ClosePictureFile (pfPicture);
NormalizeRect (&rcFrame);
```

See Also:

Functions GetMoviesError, GetMoviesStickyError

## *OpenMovieFile*

Syntax

```
OSErr OpenMovieFile (LPCSTR lpstrFileSpec, SHORT FAR
*MovieFile, int sOFlag)
```

`OpenMovieFile` opens a file containing a movie.

Parameters

*`LPCSTR`* `lpstrFileSpec`

The name of a string containing the movie file name.

*`SHORT FAR`* `*MovieFile`

The address of a reference value which will be assigned by this function, and which will be used by NewMovieFromFile and CloseMovieFile. Valid values are in the range `0x000` through `0xFFFE`. `0xFFFF` indicates and invalid value.

*`int`* `sOFlag`

An integer expressed as a standard file open flag as defined for the Windows `OpenFile` function. Movie files are normally opened as read only (use the OF_READ flag).

Return

Zero if no error condition. Non-zero if error condition. See Appendix A for error condition values. You can use GetMoviesError and GetMoviesStickyError to test for failure of this call.

Comments

QuickTime for Windows movie file names have the DOS suffix ".MOV".

To play *n* cases of the same movie simultaneously, the movie file must be opened *n* times to get *n* unique movie objects and then associate *n* movie controllers.

Example

```
MovieFile mfMovie;
Movie mMovie;
•
•
if (!OpenMovieFile ("MOVIE.MOV", &mfMovie, OF_READ))
   {
   NewMovieFromFile (&mMovie, mfMovie, NULL, NULL,
      newMovieActive, NULL);
   CloseMovieFile (mfMovie);
   }
else
   {
```

```
MessageBox (hWnd, "OpenMovieFile failure",
   "Movie Initialization", MB_OK);
}
```

See Also:

Functions NewMovieFromFile, CloseMovieFile, GetMoviesError, GetMoviesStickyError

## *OpenPictureFile*

Syntax `OSErr OpenPictureFile (LPCSTR lpstrFileSpec, PicFile FAR *pfPicture, int sOFlag)`

`OpenPictureFile` opens a file containing a picture.

Parameters *`LPCSTR`* `lpstrFileSpec`

A pointer to a string containing the picture file name.

*`PicFile FAR`* `*pfPicture`

The address of a reference value which will be assigned by this function, and which will be used by ClosePictureFile and other routines that reference picture data. Valid values range from `0x000` through `0xFFFE`. `0xFFFF` indicates and invalid value.

*`int`* `sOFlag`

An integer expressed as a standard file open flag as defined for the Windows `OpenFile` function. Picture files are normally opened as read only (use the OF_READ flag).

Return Zero if no error condition. Non-zero if error condition. See Appendix A for error condition values. You can use GetMoviesError and GetMoviesStickyError to test for failure of this call.

Comments QuickTime for Windows picture files are characterized by the DOS suffix ".PIC".

Example

```
PicFile pfPicture;
•
•
if (OpenPictureFile ("PICTURE.PIC", &pfPicture, OF_READ))
   {
   /* Inform user of failure. */
   }
```

See Also:

Functions ClosePictureFile, GetMoviesError, GetMoviesStickyError

## *PictureToDIB*

Syntax `DIBHandle PictureToDIB (PicHandle pcThePict)`

`PictureToDIB` converts a QuickTime for Windows format picture to a Windows compatible Device Independent Bitmap (DIB) format.

Parameters

*`PicHandle`* `pcThePict`

The QuickTime for Windows picture object.

Return

A handle to a Windows Device Independent Bitmap (DIB). You can use GetMoviesError and GetMoviesStickyError to test for failure.

Comments

The QuickTime for Windows format picture may be drawn directly to the screen without conversion to a Windows DIB by using the DrawPicture function. The object returned by `PictureToDIB` must be freed by the Windows `GlobalFree` function when you are through using it. It is, however, created with the `GMEM_SHARE` flag, so you can conveniently load the DIB to the Windows clipboard.

Example

```
Movie mMovie;
PicHandle phPicture;
DIBHandle hdPicture;
•
•
// Get the poster frame and convert to Windows DIB

   phPicture = GetMoviePosterPict (mMovie);
   hdPicture = PictureToDIB (phPicture);

// Put the DIB in the clipboard

   OpenClipboard (hWnd);
   EmptyClipboard ();
   SetClipboardData (cf_DIB, hdPicture);
   CloseClipboard ();
   DisposePicture (phPicture);
```

See Also:

Functions

DrawPicture, GetMoviePosterPict, GetMoviePosterTime, MCGetCurrentTime, GetMoviesError, GetMoviesStickyError

## *PrerollMovie*

Syntax

```
OSErr PrerollMovie (Movie mMovie, TimeValue tvTime, LFIXED
lfxRate)
```

`PrerollMovie` prepares a portion of a movie for playback, to enhance playback performance.

Parameters

*`Movie`* `mMovie`

The movie object.

*TimeValue* tvTime

A TimeValue specifying the starting time of the movie segment to play.

LFIXED *lfxRate*

Specifies the anticipated rate at which the movie will play. Positive values indicate forward rates, negative values reverse rates. The rate is used as a multiplier for the movie's recorded rate.

Return

Zero if no error condition. Non-zero if error condition. See Appendix A for error condition values. You can use GetMoviesError and GetMoviesStickyError to test for failure of this call.

Comments

Playback performance can be improved if PrerollMovie is called prior to playing a movie.

Example

```
Movie mMovie;
TimeValue tvTime;
LFIXED lfxRate;
•
•
PrerollMovie (mMovie, tvTime, lfxRate);
```

See Also:

Functions

GetMoviesError, GetMoviesStickyError

Data Types

TimeValue, LFIXED

## *PtInMovie*

Syntax

```
BOOL PtInMovie (Movie mMovie, POINT ptPoint)
```

PtInMovie determines whether a specified point lies in a movie.

Parameters

*Movie* mMovie

The movie object.

*POINT* ptPoint

The point to test, in window coordinates.

Return

TRUE if the point is in the movie rectangle, FALSE if not. You can use GetMoviesError and GetMoviesStickyError to test for error conditions.

Comments

The specified point must be supplied in window coordinates.

*Note: All QuickTime for Windows routines referencing a RECT or POINT assume client device coordinates.*

Example

```
Movie mMovie;
POINT ptTest;
•
•
if (PtInMovie (mMovie, ptTest))
   {
   /* Take appropriate action. */
   }
```

See Also:

Functions GetMovieBox, GetMoviesError, GetMoviesStickyError

## *QTFOURCC*

Syntax `QTFOURCC(ch0, ch1, ch2, ch3)`

`QTFOURCC` is a macro used to construct a four-character constant, normally used to extract user data from a movie.

Parameters *ch0...ch3*

The four characters to be concatenated.

Comments Each parameter must be enclosed in single quotes.

Example

```
UserData udData;
OSType osType;
•
•
osType = QTFOURCC('©','d','a','y');
osType = GetNextUserDataType (udData, osType);
```

## *QTInitialize*

Syntax `OSErr QTInitialize (LPLONG lplVersion)`

`QTInitialize` binds applications to QuickTime for Windows at run time. It must be called before any other QuickTime for Windows function.

Parameters `LPLONG lplVersion`

The address of a value that will be filled with the current QuickTime for Windows version number.

Return Zero if no error condition. Non-zero if error condition. See Appendix A for error condition values. You can use GetMoviesError and GetMoviesStickyError to test for failure of this call.

Comments This function must be called before any other QuickTime for Windows function. It is recommended that it be called before your program creates its main window. If your

program employs DLLs that make QuickTime for Windows calls, each DLL must call `QTInitialize`, preferably in the `LibMain` function. `QTInitialize` only needs to be called once during the life of your program. The return codes can be used to determine whether QuickTime for Windows is installed and if the hardware is capable of running it.

If `lplVersion` is not coded as `NULL`, `QTInitialize` fills the value it points to with the current QuickTime for Windows version: bits 31-16, reserved, always 0; bits 15-12, major release level; bits 11-8, minor release

level; bits 7-0, revision number. For example, `0x00001000L` is QuickTime for Windows version 1.0.0. A program can use this data to check if it is running under a certain QuickTime for Windows version, then react accordingly.

Example

```
LONG lVersion;
•
•
if ((QTInitialize (lVersion) != QTI_OK)
   || (lVersion < 0x00001000L))
   {
   MessageBox (hWnd, "QuickTime for Windows not loaded"
      " or wrong version present.",
      "QuickTime for Windows Initialization", MB_OK);
   return 0;
   }
```

See Also:

Functions QTTerminate, EnterMovies

## *QTTerminate*

Syntax `VOID QTTerminate (VOID)`

`QTTerminate` severs links to QuickTime for Windows.

Parameters None.

Return None.

Comments If your program uses DLLs, each must call `QTTerminate`, preferably in the `WEP` function.

Example

```
// Cut the connections to QuickTime for Windows

   QTTerminate ();
```

See Also:

Functions QTInitialize, ExitMovies

## *SetMovieActive*

Syntax `VOID SetMovieActive (Movie mMovie, BOOL bActive)`

`SetMovieActive` sets a movie's state to active or inactive.

Parameters *`Movie`* `mMovie`

The movie object whose state is to be changed.

*`BOOL`* `bActive`

`TRUE` sets the movie state to active, `FALSE` to inactive.

Return None. Use GetMoviesError and GetMoviesStickyError to test for failure of this call.

Comments An inactive movie does not receive cycles from QuickTime for Windows' internal scheduler, so it will not play. Setting a movie inactive can be used to control which one of several simultaneously playing movies will receive system resources. You can query a movie's active state using GetMovieActive.

Simply setting a movie to the active state does not affect any of its attributes, such as visibility. You have to explicitly update a window in which a movie appears if the movie is made active.

Example

```
Movie mMovie;
•
•
// Deactivate the movie

   SetMovieActive (mMovie, FALSE);

// Re-activate the movie

   SetMovieActive (mMovie, TRUE);
```

See Also:

Functions GetMovieActive, GetMoviesError, GetMoviesStickyError, MCActivate

## *SetMovieCoverProcs*

Syntax `VOID SetMovieCoverProcs (Movie mMovie, CoverProc UncoverProc, CoverProc CoverProc, LONG lRefCon)`

`SetMovieCoverProcs` sets cover and uncover procedures for your movie.

Parameters *`Movie`* `mMovie`

The movie object.

*`CoverProc`* `UncoverProc`

The address of the uncover procedure.

*CoverProc* CoverProc

The address of the cover procedure.

*LONG* lRefCon

A reference constant that is passed to the cover procedure.

Return

None. You can use GetMoviesError and GetMoviesStickyError to test for failure of this call.

Comments

This routine allows your program to perform custom processing whenever one of your movies covers a screen region or reveals a region that was previously covered. This activity is performed in cover procedures, of which there are two types: those called when your movie covers a screen region, and those called when your movie uncovers a screen region that was previously covered. The former is responsible for saving the region (you may choose to save the hidden region in an offscreen buffer).

Cover procedures called when your movie reveals a hidden screen region may redisplay the hidden region. If no uncover procedure is supplied, the default action is to paint the uncovered region with the background brush saved when the movie was created (GetClassWord, GetObject and CreateBrushIndirect). If no background brush is found, a solid white brush will be used. There is no default action if you do not supply a cover procedure.

If you compile your program using Borland *smart callbacks* or Microsoft's -GEs compiler option, or your filter function is in a dynamic link library, you do not need to use MakeProcInstance on your cover procedure address before calling SetMovieCoverProcs.

Example

```
OSErr CALLBACK __export MyCoverProc (Movie, HDC, LONG);

HWND hWnd;
Movie mMovie;
•
•
SetMovieCoverProcs (mMovie, MyCoverProc, NULL, 5879);
•
•
OSErr CALLBACK __export MyCoverProc (Movie m, HDC hdc, lID)
   {
   RECT rcClip;
   GetClipBox (hdc, &rcClip);
   FillRect (hdc, &rcClip, GetStockObject (WHITE_BRUSH));
   return 0;
   }
```

See Also:

Functions

CoverProc, GetMoviesError, GetMoviesStickyError

## *SubtractTime*

Syntax

```
VOID SubtractTime (TimeRecord FAR *lptrDst, const TimeRecord  FAR
```

```
*lptrSrc)
```

`SubtractTime` subtracts one time from another.

Parameters

TimeRecord `*lptrDst`

The address of a time record containing the first operand for the subtraction. The time record is overwritten by the result.

*`const`* TimeRecord *`FAR`* `*lptrSrc`

The address of a time record containing the second operand, which remains unmodified by the operation.

Return

None. The result is in the time record referenced by the first parameter. Use GetMoviesError and GetMoviesStickyError to test for failure.

Comments

If the time records have different time scales, `SubtractTime` converts them.

Example

```
MovieController mcController;
TimeRecord trOne, trTwo;
•
•
SubtractTime (&trOne, &trTwo);
MCDoAction (mcController, mcActionGoToTime, (LPVOID) &trOne);
```

See Also:

Functions

ConvertTimeScale, GetMovieTimeScale, AddTime, GetMoviesError, GetMoviesStickyError

MCDoAction

mcActionGoToTime

Data Types

TimeRecord, `TimeScale`

## *UpdateMovie*

Syntax

```
OSErr UpdateMovie (Movie mMovie)
```

`UpdateMovie` paints the current movie image on demand, rather than at its scheduled time.

Parameters

*`Movie`* `mMovie`

The movie object.

Return

Zero if no error condition. Non-zero if error condition. See Appendix A for error condition values. You can use `Get`MoviesError and GetMoviesStickyError to test for failure of this call.

Comments `UpdateMovie` allows you to manually refresh the current movie image.

Example

```
Movie mMovie;
•
•
UpdateMovie (mMovie);
```

See Also:

Functions GetMoviesError, GetMoviesStickyError

# B. QuickTime for Windows API - Data Structures

ImageDescription
Int64
LFIXED
OpenCPicParams
SFIXED
SoundDescription
TimeRecord

### *ImageDescription*

Description The `ImageDescription` structure contains information about a picture file.

Syntax

```
typedef struct  // Hungarian: id (ImageDescription)
  {
  LONG   idSize;
  DWORD CodecType;
  DWORD resvd1;
  WORD   resvd2;
  WORD   dataRefIndex;
  WORD   version;
  WORD   revLevel;
  DWORD vendor;
  DWORD temporalQuality;
  DWORD spatialQuality;
  WORD   width;
  WORD   height;
  LFIXED   hRes;
  LFIXED   vRes;
  DWORD dataSize;
  WORD   frameCount;
  char   name [32];
  WORD   depth;
  WORD   clutID;
} ImageDescription;
```

Fields

`idSize`

Specifies the structure size.

`CodecType`

Specifies the Codec Type:

`'rpza'` = Apple video

`'jpeg'` = Apple JPEG

`'rle '` = Apple animation

`'raw '` = Apple raw

`'smc '` = Apple graphics

`'cvid'` = Compact Video

rsvd1

Reserved, always 0.

rsvd2

Reserved, always 0.

`dataRefIndex`

Reserved, always 1.

`version`

Reserved, always 0.

`revLevel`

Reserved, always 0.

`vendor`

Reserved, always 0.

`temporalQuality`

Reserved, always 0.

`spatialQuality`

Reserved, always 0.

`width`

Specifies the Source image width in pixels.

`height`

Specifies the Source image height in pixels.

`hRes`

Specifies the horizontal resolution (e.g. 72.0).

`vRes`

Specifies the vertical resolution (e.g. 72.0).

`dataSize`

Reserved, always 0.

`frameCount`

Reserved, always 0.

`name [32]`

Specifies the compression algorithm (e.g. Animation).

`depth`

Specifies the pixel depth of the source image.

`clutID`

Reserved, always 0.

Comments

This structure is populated by QuickTime for Windows calls that request information about a picture file (for example, GetPictureInfo).

## *Int64*

Description The `Int64` structure defines a quad word for use in other structures.

Syntax

```
typedef struct        // Hungarian: qw (quad word)
  {
  LONG   dwLo;
  DWORD dwHi;
  } Int64;
```

Fields

`dwLo`

Specifies the low order double word.

`dwHi`

Specifies the high order double word.

Comments This structure is used by the TimeRecord structure.

## *LFIXED*

Description The `LFIXED` type defines a long integer where the high-order sixteen bits define a signed short integer representing an integral value and the low-order sixteen bits define an unsigned short integer representing a fractional value.

Comments `LFIXED` variables are normally used to hold movie rates in QuickTime for Windows. For example, the `LFIXED` value `0x00028000` could be used to represent a rate of 2.5.

See Also:

Functions `MAKELFIXED` (macro)

Data Types SFIXED

## *OpenCPicParams*

Description The `OpenCPicParams` structure defines the picture file header.

Syntax

```
typedef struct     // Hungarian: ocp
  {
  RECT   rect;
  LFIXED   hRes;
  LFIXED   vRes;
  WORD   wVersion;
  WORD   wReserved1;
```

```
DWORD dwReserved2;
} OpenCPicParams;
```

Fields

`rect`

Specifies a picture rectangle.

`hRes`

Specifies the horizontal resolution (e.g. 72.0).

`vRes`

Specifies the vertical resolution (e.g. 72.0).

`wVersion`

Specifies the version.

`wReserved1`

Reserved, always 0.

`dwReserved2`

Reserved, always 0.

Comments

This structure is populated by QuickTime for Windows calls that return the picture file header (for example, GetPictureFileHeader).

## *SFIXED*

Description

The `SFIXED` type defines a short integer where the high-order eight bits define a signed integer value and the low-order eight bits define an unsigned fractional value.

Comments

`SFIXED` variables are normally used to hold movie sound track volumes in QuickTime for Windows. For example, the `SFIXED` value `0x0080` could be used to represent a sound volume of 0.5.

See Also:

Functions `MAKESFIXED` (macro)

Data Types LFIXED

## *SoundDescription*

Description

The `SoundDescription` structure contains information about a movie's sound.

Syntax

```
typedef struct   // Hungarian: sd (SoundDescription)
  {
```

```
LONG   descSize;
DWORD dataFormat;
DWORD resvd1;
WORD   resvd2;
WORD   dataRefIndex;
WORD   version;
WORD   revLevel;
DWORD vendor;
WORD   numChannels;
WORD   sampleSize;
WORD   compressionID;
WORD   packetSize;
LFIXED   sampleRate;
} SoundDescription;
```

Fields

`descSize`

Specifies the structure size.

`dataFormat`

Specifies the data format (always 'raw').

`resvd1`

Reserved, always 0.

`resvd2`

Reserved, always 0.

`dataRefIndex`

Reserved, always 1.

`version`

Reserved, always 0.

`revLevel`

Reserved, always 0.

`vendor`

Reserved, always 0.

`numChannels`

Specifies the channels: 1 = mono, 2 = stereo.

`sampleSize`

Specifies the sample size: 8 = 8-bit sound, 16 = 16-bit sound.

`compressionID`

Reserved, always 0.

`packetSize`

Reserved, always 0.

`sampleRate`

Sample rate, e.g. 44100.0000 per second.

Comments — This structure is populated by QuickTime for Windows calls that request information about a movie file's sound (see GetSoundInfo).

### *TimeRecord*

Description — The `TimeRecord` structure defines a point in a movie's time coordinate system.

Syntax

```
typedef struct        // Hungarian: tr (TimeRecord)
   {
   Int64     value;
   TimeScale scale;
   TimeBase  base;
   } TimeRecord;
```

Fields

`value`

Specifies a movie time value.

`scale`

Specifies the movie's time scale.

`base`

NULL - means that the `TimeRecord` specifies a duration, or TIMEBASE_DEFALUT - means that the `TimeRecord` specifies a time, relative to the start of the movie.

Comments — The minimum `TimeValue` is 0, which is the very beginning of a movie. A `TimeValue` is expressed in time units which are related to the movie's time scale.

The time coordinate system contains a time scale scored in time units. The number of units that pass per second quantifies the scale: a time scale of 26 means that 26 units pass per second and each time unit is 1/26 of a second.

When the duration of all or part of a movie is needed, it is expressed as the length of the portion of the movie in the number of time units it contains. Particular points in a movie can be identified by a time value, which is the number of time units to that point from the beginning of the movie.

Different movies may have different time scales. Use ConvertTimeScale to compare `TimeValues` between different movies.

# Appendix A. QuickTime for Windows Error Codes

| | | |
|---|---|---|
| -2001 | `badImageDescription` | Problem with this image description. |

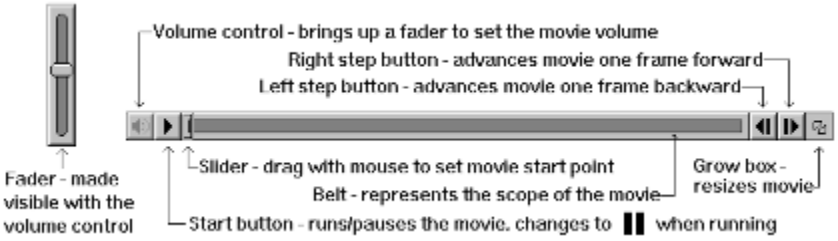
| | | |
|---|---|---|
| -2002 | badPublicMovieAtom | Movie file corrupted. |
| -2010 | invalidMovie | This movie is corrupted or invalid. |
| -2011 | invalidSampleTable | This sample table is corrupted or invalid. |
| -2014 | invalidDuration | This duration value is invalid. |
| -2015 | invalidTime | This time value is invalid. |
| -2017 | badEditList | This track's edit list is corrupted. |
| -2020 | movieToolboxUninitialized | You haven't initialized the Movie Toolbox. |
| -2021 | wffileNotFound | Cannot locate this file. |
| -2026 | userDataItemNotFound | Cannot locate this user data item. |
| -2027 | maxSizeToGrowTooSmall | Maximum size must be larger. |
| -2034 | internalQuickTimeError | Internal value. |
| -2036 | invalidRect | Specified rectangle has invalid coordinates. |
| -2039 | invalidSampleDescIndex | Sample description index value invalid. |
| -2041 | invalidSampleDescription | This sample description is invalid or corrupted. |
| -2043 | dataNotOpenForRead | Cannot read from this data source. |
| -2045 | dataAlreadyClosed | You have already closed this data source. |
| -2048 | noMovieInDataFork | Toolbox cannot find a movie in the movie file. |
| -2053 | featureUnsupported | Movie Toolbox does not support this feature. |
| -2054 | noVideoTrackInMovie | No video track found in this movie. |
| -2055 | noSoundTrackInMovie | No sound track found in this movie. |
| -2056 | soundSupportNotAvailable | Sound support unavailable. |
| -2057 | maxControllersExceeded | The limit on movie controllers has been reached. |
| -2058 | unableToCreateMCWindow | Cannot create the Movie Controller window. |
| -2059 | insufficientMemory | Memory allocation request failed. |
| -2060 | invalidUserDataHandle | Request for user data failed based on handle used. |
| -2061 | noPictureInFile | File is valid but contains no pictures. |
| -9995 | editingNotAllowed | Editing is not supported. |
| -9996 | controllerBoundsNotExact | The movie controller bounds are not exact. |
| 0 | mcEventNotHandled | Movie controller event not handled. |
| 0 | mcOK | Movie controller OK. |
| 0 | noErr | Action complete successfully. |
| 0 | QTI_OK | Initialization is OK. |
| 1 | mcEventHandled | Movie controller event handled. |
| 1 | QTI_FAIL_NOEXIST | Initialization failed, system not found. |
| 2 | QTI_FAIL_CORRUPTDLL | Corrupt DLL found at initialization. |
| 3 | QTI_FAIL_286 | Cannot initialize on a 80286 platform. |
| 4 | QTI_FAIL_WIN30 | Cannot initialize on Windows release 3.0. |
| 0x80008001 | badComponentInstance | Component instance not valid. |
| 0x80008002 | badComponentSelector | Component selector not valid. |

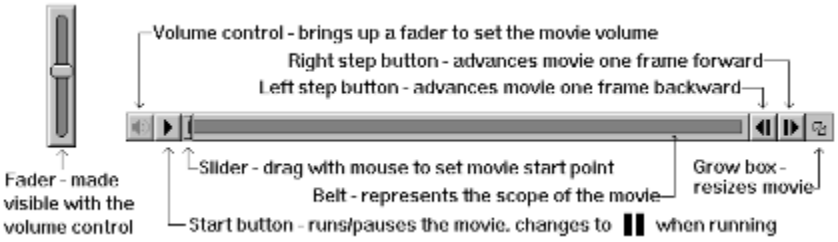
# Appendix B. Region Codes

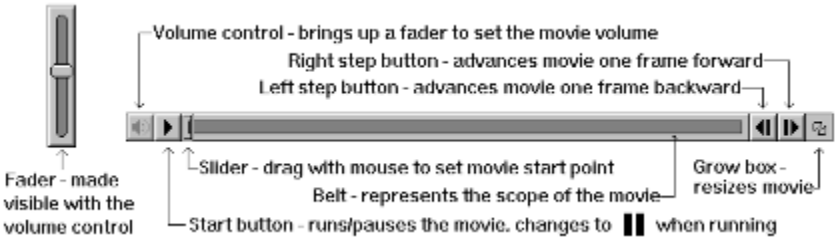
The following codes are used to identify specific languages in the function `GetUserDataText` when alternative text or multiple languages are supported. See the description of GetUserDataText for further information.

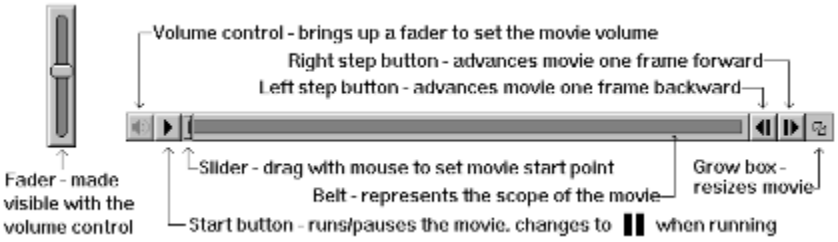
| | | | |
|---|---|---|---|
| verUS | 0 | verIceland | 21 |
| verFrance | 1 | verMalta | 22 |
| verBritian | 2 | verCyprus | 23 |
| verGermany | 3 | verTurkey | 24 |
| verItaly | 4 | verYugoCroatian | 25 |
| verNetherlands | 5 | verIndiaHindi | 33 |

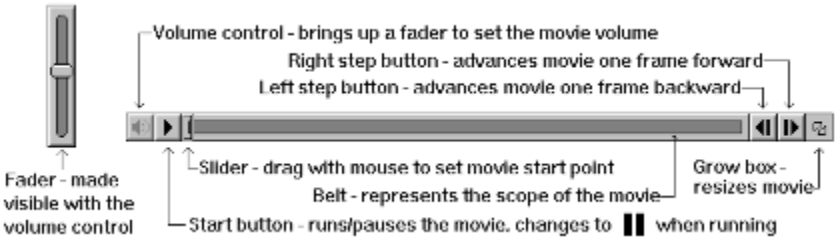
| | | | |
|---|---|---|---|
| verFrBelgiumLux | 6 | verPakistan | 34 |
| verSweden | 7 | verLithuania | 41 |
| verSpain | 8 | verPoland | 42 |
| verDenmark | 9 | verHungary | 43 |
| verPortugal | 10 | verEstonia | 44 |
| verFrCanada | 11 | verLatvia | 45 |
| verNorway | 12 | verLapland | 46 |
| verIsrael | 13 | verFaeroeIsl | 47 |
| verJapan | 14 | verIran | 48 |
| verAustralia | 15 | verRussia | 49 |
| verArabic | 16 | verIreland | 50 |
| verFinland | 17 | verKorea | 51 |
| verFrSwiss | 18 | verChina | 52 |
| verGrSwiss | 19 | verTaiwan | 53 |
| verGrverIceland | 20 | verThailand | 54 |

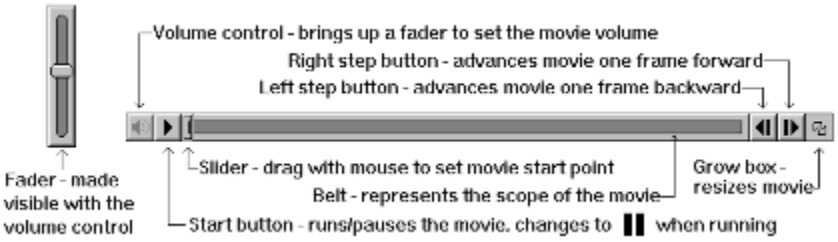


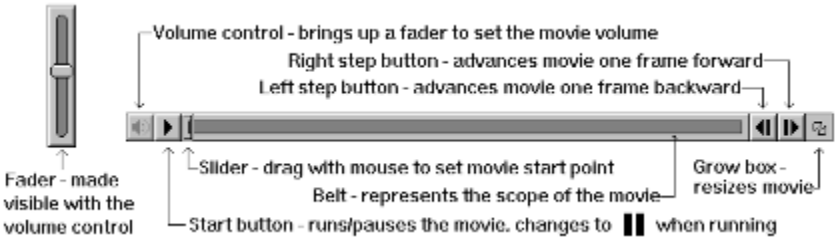

Volume control - brings up a fader to set the movie volume
Right step button - advances movie one frame forward
Left step button - advances movie one frame backward
Fader - made visible with the volume control
Slider - drag with mouse to set movie start point
Belt - represents the scope of the movie
Grow box - resizes movie
Start button - runs/pauses the movie. changes to
when running

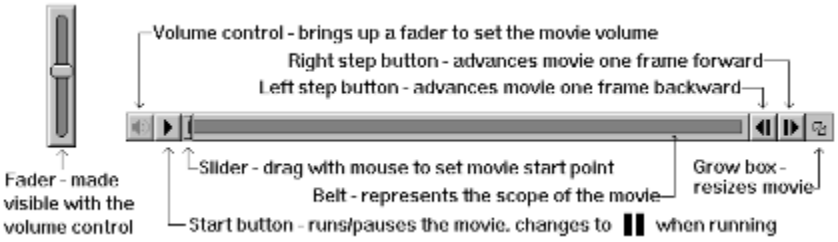

Volume control - brings up a fader to set the movie volume
Right step button - advances movie one frame forward
Left step button - advances movie one frame backward
Fader - made visible with the volume control
Slider - drag with mouse to set movie start point
Belt - represents the scope of the movie
Grow box - resizes movie
Start button - runs/pauses the movie. changes to
when running

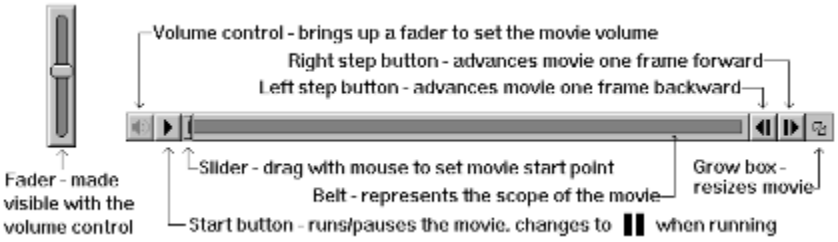

Volume control - brings up a fader to set the movie volume
Right step button - advances movie one frame forward
Left step button - advances movie one frame backward
Fader - made visible with the volume control
Slider - drag with mouse to set movie start point
Belt - represents the scope of the movie
Grow box - resizes movie
Start button - runs/pauses the movie. changes to
when running

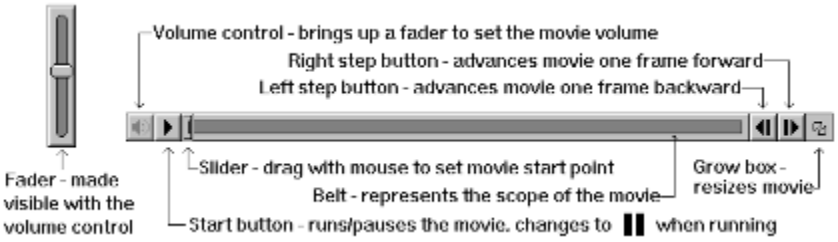

Volume control - brings up a fader to set the movie volume
Right step button - advances movie one frame forward
Left step button - advances movie one frame backward
Fader - made visible with the volume control
Slider - drag with mouse to set movie start point
Belt - represents the scope of the movie
Grow box - resizes movie
Start button - runs/pauses the movie. changes to
when running


Volume control - brings up a fader to set the movie volume
Right step button - advances movie one frame forward
Left step button - advances movie one frame backward
Fader - made visible with the volume control
Slider - drag with mouse to set movie start point
Belt - represents the scope of the movie
Grow box - resizes movie
Start button - runs/pauses the movie. changes to
when running


Volume control - brings up a fader to set the movie volume
Right step button - advances movie one frame forward
Left step button - advances movie one frame backward
Fader - made visible with the volume control
Slider - drag with mouse to set movie start point
Belt - represents the scope of the movie
Grow box - resizes movie
Start button - runs/pauses the movie. changes to
when running


Volume control - brings up a fader to set the movie volume
Right step button - advances movie one frame forward
Left step button - advances movie one frame backward
Fader - made visible with the volume control
Slider - drag with mouse to set movie start point
Belt - represents the scope of the movie
Grow box - resizes movie
Start button - runs/pauses the movie. changes to
when running


Volume control - brings up a fader to set the movie volume
Right step button - advances movie one frame forward
Left step button - advances movie one frame backward
Fader - made visible with the volume control
Slider - drag with mouse to set movie start point
Belt - represents the scope of the movie
Grow box - resizes movie
Start button - runs/pauses the movie. changes to
when running


Volume control - brings up a fader to set the movie volume
Right step button - advances movie one frame forward
Left step button - advances movie one frame backward
Fader - made visible with the volume control
Slider - drag with mouse to set movie start point
Belt - represents the scope of the movie
Grow box - resizes movie
Start button - runs/pauses the movie. changes to
when running