

## Text Media Handler Functions

---

This section describes the functions and structure associated with the text media handler, which allows you to display text in movies. You can use text media handlers to

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

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

The Movie Toolbox provides functions to help you add text samples to a track. You can use the `AddTextSample` function to add text to a media. The `AddTESample` function allows you to specify a `TextEdit` handle (which may have multiple style runs) to be added to a media. The `AddHiliteSample` function allows you to indicate highlighting for text that has just been added with the `AddTextSample` or `AddTESample` function. For more information on styled text, style runs, and `TextEdit`, see *Inside Macintosh: Text*.

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

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

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

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

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

## Movie Toolbox

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

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

The text description structure is defined as follows:

```
typedef struct TextDescription {
    long          size;           /* total size of this text
                                description structure */
    long          type;           /* type of data in this
                                structure such as
                                'text' */
    long          resvd1;         /* reserved for use by
                                Apple--set to 0 */
    long          resvd2;         /* reserved for use by
                                Apple--set to 0 */
    short         dataRefIndex;   /* index to data references */
    long          displayFlags;   /* display flags for text */
    long          textJustification; /* text justification flags */
    RGBColor      gColor;         /* background color */
    Rect          defaultTextBox; /* location of the text within
                                track bounds */
    ScrpSTElement defaultStyle;   /* default style--
                                TextEdit structure */
} TextDescription, *TextDescriptionPtr, **TextDescriptionHandle;
```

**Field descriptions**

size	Defines the total size of this text description structure.
type	Indicates the type (data type 'text').
resvd1	Reserved for use by Apple. This field must be set to 0.
resvd2	Reserved for use by Apple. This field must be set to 0.
displayFlags	Contains the flags that specify how the text is to be displayed.
textJustification	Contains the constant that specifies how the text is to be aligned.
bgColor	Specifies the background color for the text display.
defaultTextBox	Indicates the location of the text within track boundaries.
defaultStyle	Provides a TextEdit data structure (defined by the ScrpSTElement data type) that specifies the default style for the text display.

## Movie Toolbox

The `AddTextSample`, `AddTESample`, and `AddHiliteSample` functions described in the sections that follow convert text into the text media format and add it to the media. To use these functions, you need to

- create a text track and media
- call the `BeginMediaEdits` function
- call the `AddTextSample`, `AddTESample`, or `AddHiliteSample` function, as appropriate
- call the `EndMediaEdits` function
- call the `InsertMediaIntoTrack` function

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

- If you try to paste text, the text is inserted at the current position. The text import component tries to find an existing text track that fits the text.
- If no text tracks exist and there is an insertion operation, the newly created text track has the same position and size as the movie box.
- If there is an addition operation (using the Shift key), the new track is added below the movie at a height that fits the text.
- If a text track exists but the text does not fit, a new text track with sufficient height to accommodate the text is created in the same location as the existing one.
- If you hold down the Option key when you paste, the text is added in parallel at some default duration.
- If you hold down both the Option and Shift keys, the duration of the text is determined by the length of the current selection.
- If style information is on the Clipboard, it is used; otherwise, the text appears in the default 12-point application font, centered, in white on a black background.

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

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

For details on the movie import and export components, see *Inside Macintosh: QuickTime Components*.

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

You can use the `SetTextProc` function (also described in this section) to specify a customized function whenever a new text sample is added to a movie. The application-defined text function `MyTextProc` is described in “Text Functions” on page 2-364.

## AddTextSample

---

The `AddTextSample` function adds a single block of styled text to an existing media.

```
pascal ComponentResult AddTextSample (MediaHandler mh, Ptr text,
                                     unsigned long size,
                                     short fontNum,
                                     short fontSize,
                                     Style textFace,
                                     RGBColor *textColor,
                                     RGBColor *backColor,
                                     short textJustification,
                                     Rect *textBox,
                                     long displayFlags,
                                     TimeValue scrollDelay,
                                     short hiliteStart,
                                     short hiliteEnd,
                                     RGBColor *rgbHiliteColor,
                                     TimeValue duration,
                                     TimeValue *sampleTime);
```

mh	Specifies the media handler for the text media obtained by the <code>GetMediaHandler</code> function.
text	Contains a pointer to a block of text.
size	Indicates the size of the text block (in bytes).
fontNum	Indicates the number for the font in which to display the text.
fontSize	Indicates the size of the font.
textFace	Indicates the typeface or style of the text (that is, bold, italic, and so on).

## Movie Toolbox

<code>textColor</code>	Contains a pointer to an RGB color structure specifying the color of the text.
<code>backColor</code>	Contains a pointer to an RGB color structure specifying the text background color.
<code>textJustification</code>	Indicates the justification of the text. The following constants are available: <code>teFlushDefault</code> , <code>teCenter</code> , <code>teFlushRight</code> , or <code>teFlushLeft</code> . See <i>Inside Macintosh: Text</i> for details on these constants and on text alignment.
<code>textBox</code>	Contains a pointer to the box within which the text is to be displayed. The box is relative to the track bounds.
<code>displayFlags</code>	Contains the text display flags.
<code>dfDontDisplay</code>	Does not display the specified sample.
<code>dfDontAutoScale</code>	Does not scale the text if the track bounds increase.
<code>dfClipToTextBox</code>	Clips to just the text box. (This is useful if the text overlays the video.)
<code>dfShrinkTextBoxToFit</code>	Recalculates size of the <code>textBox</code> parameter to just fit the given text and stores this rectangle with the text data.
<code>dfScrollIn</code>	Scrolls the text in until the last of the text is in view. This flag is associated with the <code>scrollDelay</code> parameter.
<code>dfScrollOut</code>	Scrolls text out until the last of the text is out of view. This flag is associated with the <code>scrollDelay</code> parameter. If both <code>dfScrollIn</code> and <code>dfScrollOut</code> are set, the text is scrolled in, then out.
<code>dfHorizScroll</code>	Scrolls a single line of text horizontally. If the <code>dfHorizScroll</code> flag is not set, then the scrolling is vertical.
<code>dfReverseScroll</code>	If set, scrolls vertically down, rather than up. If not set, horizontal scrolling proceeds toward the left rather than toward the right.
<code>scrollDelay</code>	Indicates the delay in scrolling associated with setting the <code>dfScrollIn</code> and <code>dfScrollOut</code> display flags. If the value of the <code>scrollDelay</code> parameter is greater than 0 and the <code>dfScrollIn</code> flag is set, the text pauses when it has scrolled all the way in for the amount of time specified

## Movie Toolbox

by `scrollDelay`. If the `dfScrollOut` flag is set, the pause occurs first before the text scrolls out. If both these flags are set, the pause occurs at the midpoint between scrolling in and scrolling out.

`hiliteStart`

Specifies the beginning of the text to be highlighted.

`hiliteEnd`

Specifies the end of the text to be highlighted. If the `hiliteEnd` parameter is greater than the `hiliteStart` parameter, then the text is highlighted from the selection specified by `hiliteStart` to `hiliteEnd`. To specify additional highlighting, you can use the `AddHiliteSample` function, described on page 2-297.

`rgbHiliteColor`

Contains a pointer to the RGB color for highlighting. If this parameter is not `nil`, then the specified color is used when highlighting the text indicated by the `hiliteStart` and `hiliteEnd` parameters. Otherwise, the default system highlighting is used.

`duration`

Specifies how long the text sample should last. This duration is expressed in the media's time base.

`sampleTime`

Contains a pointer to a `TimeValue` structure. The actual media time at which the sample was added is returned here.

## ERROR CODES

`invalidMedia`    -2008    This media is corrupted or invalid

File Manager errors

Memory Manager errors

## AddTESample

---

The `AddTESample` function allows you to specify a `TextEdit` handle (which may contain multiple style runs) to be added to the specified media.

```
pascal ComponentResult AddTESample (MediaHandler mh, TEHandle hTE,
                                     RGBColor *backColor,
                                     short textJustification,
                                     Rect *textBox,
                                     long displayFlags,
                                     TimeValue scrollDelay,
                                     short hiliteStart,
                                     short hiliteEnd,
                                     RGBColor *rgbHiliteColor,
                                     TimeValue duration,
                                     TimeValue *sampleTime);
```

## Movie Toolbox

<code>mh</code>	Specifies the media handler for the text media obtained by the <code>GetMediaHandler</code> function.
<code>hTE</code>	A handle to a styled <code>TextEdit</code> structure.
<code>backColor</code>	Contains a pointer to an RGB color structure specifying the text background color.
<code>textJustification</code>	Indicates the justification of the text. The following constants are available: <code>teFlushDefault</code> , <code>teCenter</code> , <code>teFlushRight</code> , or <code>teFlushLeft</code> . See <i>Inside Macintosh: Text</i> for details on these constants and on text alignment.
<code>textBox</code>	Contains a pointer to the box within which the text is to be displayed. The box is relative to the track bounds.
<code>displayFlags</code>	Contains the text display flags.
<code>dfDontDisplay</code>	Does not display the specified sample.
<code>dfDontAutoScale</code>	Does not scale the text if the track bounds increase.
<code>dfClipToTextBox</code>	Clips to the text box only. (This is useful if the text overlays the video.)
<code>dfShrinkTextBoxToFit</code>	Recalculates size of the <code>textBox</code> parameter to just fit the given text and stores this rectangle with the text data.
<code>dfScrollIn</code>	Scrolls the text in until the last of the text is in view.
<code>dfScrollOut</code>	Scrolls text out until the last of the text is out of view. If both <code>dfScrollIn</code> and <code>dfScrollOut</code> are set, the text is scrolled in, then out.
<code>dfHorizScroll</code>	Scrolls a single line of text horizontally. If the <code>dfHorizScroll</code> flag is not set, then the scrolling is vertical.
<code>dfReverseScroll</code>	If set, scrolls vertically down, rather than up. If not set, horizontal scrolling proceeds toward the left rather than toward the right.
<code>scrollDelay</code>	Indicates the delay in scrolling associated with the setting of the <code>dfScrollIn</code> and <code>dfScrollOut</code> display flags. If the value of the <code>scrollDelay</code> parameter is greater than 0 and the <code>dfScrollIn</code> flag is set, the text pauses when it has scrolled all the way in for the amount of time specified by <code>scrollDelay</code> . If the <code>dfScrollOut</code> flag is set, the pause occurs first before the text scrolls out. If both these flags are set, the pause occurs at the midpoint between scrolling in and scrolling out.

## Movie Toolbox

`hiliteStart`

Specifies the beginning of the text to be highlighted.

`hiliteEnd`

Specifies the end of the text to be highlighted. If the `hiliteEnd` parameter is greater than the `hiliteStart` parameter, then the text is highlighted from the selection specified by `hiliteStart` to `hiliteEnd`. To specify additional highlighting, you can use the `AddHiliteSample` function, described in the next section.

`rgbHiliteColor`

Contains a pointer to the RGB color for highlighting. If this parameter is not `nil`, then the specified color is used when highlighting the text indicated by the `hiliteStart` and `hiliteEnd` parameters. Otherwise, the default system highlight color is used.

`duration`

Specifies how long the text sample should last. This duration is expressed in the media's time base.

`sampleTime`

Contains a pointer to a `TimeValue` structure. The actual media time at which the sample was added is returned here.

## ERROR CODES

`invalidMedia`    -2008    This media is corrupted or invalid

File Manager errors

Memory Manager errors

## AddHiliteSample

---

The `AddHiliteSample` function provides dynamic highlighting of text.

```
pascal ComponentResult AddHiliteSample (MediaHandler mh,
                                         short hiliteStart,
                                         short hiliteEnd,
                                         RGBColor *rgbHiliteColor,
                                         TimeValue duration,
                                         TimeValue *sampleTime)
```

`mh`

Specifies the media handler for the text media obtained by the `GetMediaHandler` function.

`hiliteStart`

Indicates the beginning of the text to be highlighted.

`hiliteEnd`

Indicates the ending of the text to be highlighted. If the value of the `hiliteStart` parameter equals that of the `hiliteEnd` parameter, then no text is highlighted (that is, highlighting is turned off for the duration of the specified sample).



## Movie Toolbox

`rgbHiliteColor`

Contains a pointer to the RGB color for highlighting. If this parameter is not `nil`, then the specified color is used when highlighting the text indicated by the `hiliteStart` and `hiliteEnd` parameters. Otherwise, the default system highlight color is used.

`duration` Specifies how long the text sample should last. This duration is expressed in the media's time base.

`sampleTime`

Contains a pointer to a `TimeValue` structure. The actual media time at which the sample was added is returned here.

## DESCRIPTION

The `AddHiliteSample` function essentially extends the duration of the text that has just been added, using the highlighting indicated by the `hiliteStart` and `hiliteEnd` parameters. You must call the `AddHiliteSample` function after calling `AddTextSample` or `AddTESample`. Since `AddHiliteSample` uses the concept of difference frames, the highlighted samples must immediately follow their associated text samples.

## ERROR CODES

`invalidMedia`    -2008    This media is corrupted or invalid  
 File Manager errors  
 Memory Manager errors

**FindNextText**

---

The `FindNextText` function searches for text with a specified media handler starting at a given time.

```
pascal ComponentResult FindNextText (MediaHandler mh,
                                     Ptr text, long size,
                                     short findFlags,
                                     TimeValue startTime,
                                     TimeValue *foundTime,
                                     TimeValue *foundDuration,
                                     long *offset);
```

## Movie Toolbox

<code>mh</code>	Specifies the media handler for the text media obtained by the <code>GetMediaHandler</code> function.
<code>text</code>	Points to the text to be found.
<code>size</code>	Specifies the length of the text to be found.
<code>findFlags</code>	Specifies the conditions of the search. The following flags are available: <ul style="list-style-type: none"> <li><code>findTextEdgeOK</code> Finds sample at the given start time.</li> <li><code>findTextCaseSensitive</code> Conducts a case-sensitive search for the text.</li> <li><code>findTextReverseSearch</code> Searches backward for the text.</li> <li><code>findTextUseOffset</code> Searches beginning from the value pointed to by the <code>offset</code> parameter.</li> <li><code>findTextWrapAround</code> Conducts a wraparound search when the end or the beginning of the text is reached.</li> </ul>
<code>startTime</code>	Indicates the time (expressed in the movie time scale) at which to begin the search.
<code>foundTime</code>	Contains a pointer to the movie time at which the text sample is found if the search is successful. Otherwise, it returns -1.
<code>foundDuration</code>	Contains a pointer to the duration of the sample (in the movie time scale) that is found if the search is successful.
<code>offset</code>	Contains a pointer to the offset of the found text from the beginning of the text portion of the sample.

## DESCRIPTION

If the text sample is found, `FindNextText` returns the movie time at which it was located, the duration of the text sample, and its offset from the beginning of the text portion of the media sample.

## ERROR CODES

<code>invalidMedia</code>	-2008	This media is corrupted or invalid
File Manager errors		
Memory Manager errors		

## HiliteTextSample

---

When you call the `HiliteTextSample` function with a given text media handler, your application can specify selected text to be highlighted.

```
pascal ComponentResult HiliteTextSample (MediaHandler mh,
                                          TimeValue sampleTime,
                                          short hiliteStart,
                                          short hiliteEnd
                                          RGBColor *rgbHiliteColor);
```

<code>mh</code>	Specifies the media handler for the text media obtained by the <code>GetMediaHandler</code> function.
<code>sampleTime</code>	Indicates a sample time (in the movie time scale) for the text to be highlighted. To turn off the highlighting in the text, pass a value of -1.
<code>hiliteStart</code>	Specifies the beginning of the text to be highlighted.
<code>hiliteEnd</code>	Specifies the end of the text to be highlighted.
<code>rgbHiliteColor</code>	Contains a pointer to the RGB color for highlighting. If this parameter is not nil, then the specified color is used when highlighting the text indicated by the <code>hiliteStart</code> and <code>hiliteEnd</code> parameters. Otherwise, the default system highlight color is used.

### DESCRIPTION

The `HiliteTextSample` function overrides any highlighting information that may already be in the specified text.

### ERROR CODES

None

### SEE ALSO

The `HiliteTextSample` function is useful when used in conjunction with the `FindNextText` function, described in the previous section.

## SetTextProc

---

Your application can use the `SetTextProc` function to specify a customized function that is to be called whenever a text sample is displayed in a movie.

```
pascal ComponentResult SetTextProc (MediaHandler mh,
                                    TextMediaProcPtr TextProc,
                                    long refcon);
```

- |                       |  |
|-----------------------|--|
| <code>mh</code>       | Indicates the media handler for the text media obtained by the <code>GetMediaHandler</code> function.              |
| <code>TextProc</code> | Points to the address of your customized function.   |
| <code>refcon</code>   | Indicates a reference constant that will be passed to your function. Set this parameter to 0 if you don't need it. |
- The format of your customized text function is

```
pascal OSErr MyTextProc (Handle theText,
                          Movie theMovie,
                          short *displayFlag,
                          long refcon);
```

See “Text Functions” on page 2-364 for details on the parameters.

### ERROR CODES

None

## Functions for Creating File Previews

---

The Movie Toolbox provides two functions that allow you to create file previews. File previews contain information that gives the user an idea of a file's contents without opening the file. Typically, a file's preview is a small PICT image (called a *thumbnail*), but previews may also contain other types of information that is appropriate to the type of file being considered. For example, a text file's preview might tell the user when the file was created and what it discusses. For more information about file previews and how to display them, see “Previewing Files” on page 2-65.

### Note

The `MakeFilePreview` and `AddFilePreview` functions documented in this section are not listed in the MPW `Movies.h` interface file; rather, they appear in the MPW `ImageCompression.h` interface file. ♦

You can use the `MakeFilePreview` function to create a preview for a file. The `AddFilePreview` function allows you to add a preview that you have created to a file.

## MakeFilePreview

---

The `MakeFilePreview` function creates a preview for a file. You should create a preview whenever you save a movie. You specify the file by supplying a reference to its resource file. You must have opened this resource file with write permission.

```
pascal OSErr MakeFilePreview (short resRefNum,
                              ProgressProcRecordPtr progress);
```

**resRefNum** Specifies the resource file for this operation. You must have opened this resource file with write permission. If there is a preview in the specified file, the Movie Toolbox replaces that preview with a new one.

**progress** Points to a progress function. During the process of creating the preview, the Movie Toolbox may occasionally call a function you provide in order to report its progress. You can then use this information to keep the user informed.

Set this parameter to `-1` to use the default progress function. If you specify a progress function, it must comply with the interface defined for Image Compression Manager progress functions (see the chapter “Image Compression Manager” in this book for more information). Set this parameter to `nil` to prevent the Movie Toolbox from calling a progress function. (For details on application-defined progress functions, see “Progress Functions,” which begins on page 2-354.)

### DESCRIPTION

If there is a preview in the specified file, the Movie Toolbox replaces that preview with a new one.

### ERROR CODES

`paramErr`    `-50`    Invalid parameter specified

File Manager errors

Memory Manager errors

Resource Manager errors

## AddFilePreview

---

The `AddFilePreview` function allows you to add a preview to a file. You must have created the preview data yourself. If the specified file already has a preview defined, the `AddFilePreview` function replaces it with the new preview.

```
pascal OSErr AddFilePreview (short resRefNum, OSType previewType,  
                             Handle previewData);
```

**resRefNum** Specifies the resource file for this operation. You must have opened this resource file with write permission. If there is a preview in the specified file, the Movie Toolbox replaces that preview with a new one.

**previewType** Specifies the resource type to be assigned to the preview. This type should correspond to the type of data stored in the preview. For example, if you have created a QuickDraw picture that you want to use as a preview for a file, you should set the `previewType` parameter to `PICT`.

**previewData** Contains a handle to the preview data. For example, if the preview data is a picture, you would provide a picture handle.

### DESCRIPTION

If you pass 0 for the `previewType` and `previewData` parameters, the file preview is removed.

### ERROR CODES

File Manager errors  
Memory Manager errors  
Resource Manager errors

### SEE ALSO

You can use the `MakeFilePreview` function, described in the previous section, to create a new preview for a file.

## Functions for Displaying File Previews

The following section describes four functions that let you display file previews.

The Movie Toolbox provides two functions that allow you to display file previews in an Open dialog box in System 6 using standard file reply structures: `SFGetFilePreview` and `SFPGetFilePreview`. The Movie Toolbox also supplies two new functions that allow you to display file previews in an Open dialog box in System 7 using standard file reply structures: `StandardGetFilePreview` and `CustomGetFilePreview`.

- The `SFGetFilePreview` function corresponds to the File Manager's `SFGetFile` routine. This function is the preferred function for creating a file preview and works with either System 7 or System 6.
- The `SFPGetFilePreview` function corresponds to the File Manager's `SFPGetFile` routine.
- The `StandardGetFilePreview` function corresponds to the File Manager's `StandardGetFile` routine.
- The `CustomGetFilePreview` function corresponds to the File Manager's `CustomGetFile` routine. This function is available only in System 7.

All of these functions take the same parameters as their existing counterparts with the addition of a `where` parameter that allows you to specify the location of the upper-left corner of the dialog box. See *Inside Macintosh: Files* for information on the `SFGetFile`, `SFPGetFile`, `StandardGetFile`, and `CustomGetFile` routines.

The `SFGetFilePreview`, `SFPGetFilePreview`, `StandardGetFilePreview`, and `CustomGetFilePreview` functions allow the user to automatically convert files to movies if your application requests movies. If there is a file that can be converted into a movie file using a movie import component, then the file is shown in the Standard File dialog box in addition to any movies. When the user selects the file, the Open button changes to a Convert button. Figure 2-41 provides an example of this dialog box.

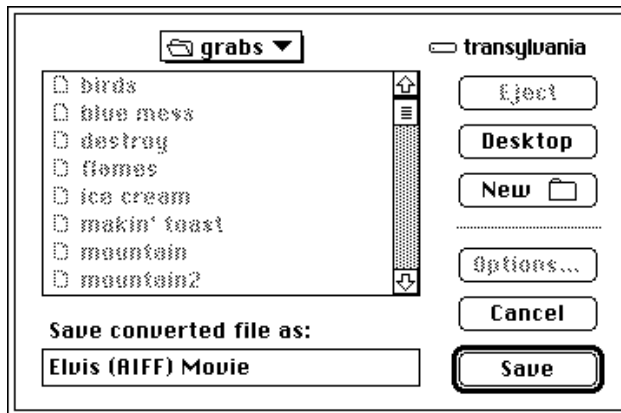
**Figure 2-41** Dialog box showing automatic file-to-movie conversion option



## Movie Toolbox

Choosing Convert displays a dialog box that allows the user to choose where the converted file should be saved. Figure 2-42 shows this dialog box.

**Figure 2-42** Dialog box for saving a movie converted from a file



When conversion is complete, the converted file is returned to the calling application as the movie that the user chose. If you want to disable automatic file conversion in your application, you must write a file filter function and pass it to the file preview display function you are using. Your file filter function must call the File Manager's `FSpGetFileInfo` function on each file that is passed to it to determine its actual file type. If the File System parameter block pointer passed to your file filter function indicates that the file type is `'Moov'`, and the actual type returned by `FSpGetFileInfo` is not `'Moov'`, then the file filter function will convert this file. If you do not wish a file to be displayed as a candidate for conversion, your file filter function should return a value of `true` when it is called for that file.

See “File Filter Functions” beginning on page 2-360 for comprehensive details on the interaction of application-defined file filter functions with the file preview display functions. For information of `FSpGetFileInfo`, see *Inside Macintosh: Files*.

#### Note

The functions described in this section do not appear in the MPW interface file `Movies.h`; rather, they are listed in `ImageCompression.h`. ♦



## SFGetFilePreview

---

The `SFGetFilePreview` function allows you to display file previews in an Open dialog box using a standard file reply structure. This is the preferred function for displaying a file preview and it works with either System 7 or System 6.

```
pascal void SFGetFilePreview (Point where,
                              ConstStr255Param prompt,
                              FileFilterProcPtr fileFilter,
                              short numTypes, SFTypelist typeList,
                              DlgHookProcPtr dlgHook,
                              SFReply *reply);
```

**where** Specifies the location of the upper-left corner of the dialog box in global coordinates. If you set this point to `(-1, -1)`, the Movie Toolbox centers the dialog box on the main screen. If you set this point to `(-2, -2)`, the Movie Toolbox centers the dialog box on the screen that has the best display characteristics.

**prompt** This parameter is ignored; it is included for historical reasons only.

**fileFilter** Points to a function that filters the files that are displayed to the user in the dialog box. This is an optional function provided by your application; if you do not want to supply a filter function, set this parameter to `nil`. The `SFGetFilePreview` function uses this parameter along with the `numTypes` and `typeList` parameters to determine which files appear in the dialog box.

If this parameter is not `nil`, `SFGetFilePreview` calls the function for each file to determine whether to display the file to the user. The `SFGetFilePreview` function supplies you with the information it receives from the File Manager's `GetFileInfo` routine (see *Inside Macintosh: Files* for more information). Your function returns a Boolean value indicating whether to display the file. Set the Boolean value to `false` to cause the file to be displayed.

Your function must provide the following interface:

```
pascal Boolean MyFileFilter (ParmBlkPtr parmBlock);
```

See "File Filter Functions" on page 2-360 for details.

**numTypes** Specifies the number of file types in the array specified by the `typeList` parameter (a number between 1 and 4). Set this parameter to `-1` to display all files.

## Movie Toolbox

**typeList** Specifies an array of file types to be displayed to the user. The `SFGetFilePreview` function only displays files whose type matches an entry in this array (unless you set the `numTypes` parameter to `-1`; in this case, the function displays all files to the user). The `SFTypeList` data type is defined as follows:

```
typedef OSType SFTypeList[4];
```

**dlgHook** Specifies a pointer to a custom dialog function. You can use this parameter to support a custom dialog box function you have supplied. If you are not supplying a custom dialog box function, set this parameter to `nil`. Your custom dialog function must present the following interface:

```
pascal short MyDlgHook (short item,
                        DialogPtr theDialog,
                        Ptr myDataPtr);
```

For more information about using custom dialog box functions with the `SFGetFilePreview` function, see “Custom Dialog Functions” on page 2-360.

**reply** Contains a pointer to a standard file reply structure that is to receive information about the user’s selection. See *Inside Macintosh: Files* for more information about reply structures.

## DESCRIPTION

The `SFGetFilePreview` function presents an Open dialog box to the user and allows the user to view file previews during the dialog. This function corresponds to the File Manager’s `SFGetFile` routine. See *Inside Macintosh: Files* for a complete description of the `SFGetFile` routine.

The `SFGetFilePreview` function takes the same parameters as its existing counterpart with the addition of a `where` parameter that allows you to specify the location of the dialog box.

The `SFGetFilePreview` function automatically converts files to movies if your application requests movies. If a file could be converted into a movie file using a movie import component, then the file is shown in the Standard File dialog box. See Figure 2-41 on page 2-304 for the dialog box with an automatic file-to-movie conversion option and Figure 2-42 on page 2-305 for the dialog box for saving a movie converted from a file.

**Note**

The `SFGetFilePreview` function does not appear in the MPW interface file `Movies.h`; rather, it’s listed in `ImageCompression.h`. ♦

## SFPGetFilePreview

---

The `SFPGetFilePreview` function allows you to display file previews in an Open dialog box using a standard file reply structure. This function differs from `SFGetFilePreview` in that you can provide a custom dialog box with any resource type and you can specify a modal-dialog filter function that allows you to gain greater control over the user interface.

```
pascal void SFPGetFilePreview (Point where,
                               ConstStr255Param prompt,
                               FileFilterProcPtr fileFilter,
                               short numTypes,
                               SFTYPEList typeList,
                               DlgHookProcPtr dlgHook,
                               SFReply *reply, short dlgID,
                               ModalFilterProcPtr filterProc);
```

**where** Specifies the location of the upper-left corner of the dialog box in global coordinates. If you set this point to `(-1, -1)`, the Movie Toolbox centers the dialog box on the main screen. If you set this point to `(-2, -2)`, the Movie Toolbox centers the dialog box on the screen that has the best display characteristics.

**prompt** This parameter is ignored; it is included for historical reasons only.

**fileFilter** Points to a function that filters the files that are displayed to the user in the dialog box. This is an optional function provided by your application; if you do not want to supply a filter function, set this parameter to `nil`. The `SFPGetFilePreview` function uses this parameter along with the `numTypes` and `typeList` parameters to determine which files appear in the dialog box.

If this parameter is not `nil`, `SFPGetFilePreview` calls the function for each file to determine whether to display the file to the user. The `SFPGetFilePreview` function supplies you with the information it receives from the File Manager's `GetFileInfo` routine (see *Inside Macintosh: Files* for more information). Your function returns a Boolean value indicating whether to display the file. Set the Boolean value to `false` to cause the file to be displayed. See "File Filter Functions," which begins on page 2-360, for details on file filter functions.

Your function must provide the following interface:

```
pascal Boolean MyFileFilter (ParmBlkPtr parmBlock);
```

## Movie Toolbox

- numTypes** Specifies the number of file types in the array specified by the `typeList` parameter. Specify a number between 1 and 4. Set this parameter to -1 to display all files.
- typeList** Specifies an array of file types to be displayed to the user. The `SFGetFilePreview` function only displays files whose type matches an entry in this array (unless you set the `numTypes` parameter to -1; in this case, the function displays all files to the user). The `SFTypeList` data type is defined as follows:

```
typedef OSType SFTypeList[4];
```

- dlgHook** Points to a custom dialog box function. You can use this parameter to support a custom dialog box function you have supplied by specifying a dialog template resource in your resource file (the dialog template's resource type must be set to 'DLOG'; you must also supply an item list in a 'DITL' resource). You specify the dialog template's resource ID with the `dlgID` parameter. If you are not supplying a custom dialog function in this manner, set this parameter to `nil`.

Your custom dialog box function must present the following interface:

```
pascal short MyDlgHook (short item,
                        DialogPtr theDialog,
                        Ptr myDataPtr);
```

- See "Custom Dialog Functions" on page 2-360 for more information on using custom dialog functions with the `SFGetFilePreview` function.
- reply** Contains a pointer to a standard file reply structure that is to receive information about the user's selection. See *Inside Macintosh: Files* for more information about reply structures.
- dlgID** Specifies the resource ID of your custom dialog template. You can use this parameter to specify a custom dialog template resource that has a resource type that differs from the standard value. Set this parameter to 0 to use the standard template.

- filterProc** Points to your modal-dialog filter function. This function gives you greater control over the interface presented to the user. Your modal-dialog filter function must present the following interface:

```
pascal Boolean MyModalFilter (DialogPtr theDialog,
                             EventRecord* theEvent,
                             short itemHit,
                             Ptr myDataPtr);
```

See "Modal-Dialog Filter Functions" beginning on page 2-362 for details.

**DESCRIPTION**

The `SFPGetFilePreview` function presents an Open dialog box to the user and allows the user to view file previews during the dialog. This function corresponds to the File Manager's `SFPGetFile` routine. The `SFPGetFilePreview` function takes the same parameters as its existing counterpart with the addition of a `where` parameter that allows you to specify the location of the dialog box. See *Inside Macintosh: Files* for a complete description of the `SFPGetFile` routine and for more information about the parameters to this function.

The `SFPGetFilePreview` function automatically converts files to movies if your application requests movies. If a file could be converted into a movie file using a movie import component, then the file is shown in the Standard File dialog box. See Figure 2-41 on page 2-304 for the dialog box with an automatic file-to-movie conversion option and Figure 2-42 on page 2-305 for the dialog box for saving a movie converted from a file.

**Note**

The `SFPGetFilePreview` function does not appear in the MPW interface file `Movies.h`; rather, it's listed in `ImageCompression.h`. ♦

## StandardGetFilePreview

---

The `SFPGetFilePreview` function allows you to display file previews in an Open dialog box using a standard file reply structure.

```
pascal void StandardGetFilePreview (FileFilterProcPtr fileFilter,
                                   short numTypes,
                                   SFTYPEList typeList,
                                   StandardFileReply *reply);
```

**fileFilter**

Points to a function that filters the files that are displayed to the user in the dialog box. This is an optional function provided by your application; if you do not want to supply a filter function, set this parameter to `nil`. The `StandardGetFilePreview` function uses this parameter along with the `numTypes` and `typeList` parameters to determine which files appear in the dialog box.

If this parameter is not `nil`, `StandardGetFilePreview` calls the function for each file to determine whether to display the file to the user. The `StandardGetFilePreview` function supplies you with information identifying the file (see *Inside Macintosh: Files* for

more information about the format of this parameter data). Your function returns a Boolean value indicating whether to display the file. Set the Boolean value to `false` to cause the file to be displayed.

Your function must provide the following interface:

```
pascal Boolean MyFileFilter (ParmBlkPtr parmBlock);
```

**numTypes** Specifies the number of file types in the array specified by the `typeList` parameter (a number between 1 and 4). Set this parameter to `-1` to display all files.

**typeList** Specifies an array of file types to be displayed to the user. The `StandardGetFilePreview` function only displays files whose type matches an entry in this array (unless you set the `numTypes` parameter to `-1`; in this case, the function displays all files to the user). The `SFTypeList` data type is defined as follows:

```
typedef OSType SFTypeList[4];
```

**reply** Contains a pointer to a reply structure that is to receive information about the user's selection. See *Inside Macintosh: Files* for more information about reply structures.

#### DESCRIPTION

The `StandardGetFilePreview` function presents an Open dialog box to the user and allows the user to view file previews. This function corresponds to the File Manager's `StandardGetFile` routine. See *Inside Macintosh: Files* for a comprehensive description of that routine and for more information about the parameters to this function. The `StandardGetFilePreview` function takes the same parameters as its existing counterpart with the addition of a `where` parameter that allows you to specify the location of the dialog box.

The `StandardGetFilePreview` function automatically converts files to movies if your application requests movies. If a file could be converted into a movie file using a movie import component, then the file is shown in the Standard File dialog box. See Figure 2-41 on page 2-304 for the dialog box with an automatic file-to-movie conversion option and Figure 2-42 on page 2-305 for the dialog box for saving a movie converted from a file.

#### Note

The `StandardGetFilePreview` function does not appear in the MPW interface file `Movies.h`; rather, it's listed in `ImageCompression.h`. ♦

## CustomGetFilePreview

---

The CustomGetFilePreview function presents an Open dialog box to the user and allows the user to view file previews. This function differs from StandardGetFilePreview in that you can provide a custom dialog template and functions to support your template.

### Note

The CustomGetFilePreview function is available only in System 7. ♦

```
pascal void CustomGetFilePreview (FileFilterYDProcPtr fileFilter,
                                short numTypes, SFTYPEList
                                typeList, StandardFileReply
                                *reply, short dlgID,
                                Point where,
                                DlgHookYDProcPtr dlgHook,
                                ModalFilterYDProcPtr filterProc,
                                short *activeList,
                                ActivateYDProcPtr activateProc,
                                void *yourDataPtr);
```

### fileFilter

Points to a function that filters the files that are displayed to the user in the dialog box. This is an optional function provided by your application; if you do not want to supply a filter function, set this parameter to nil. The CustomGetFilePreview function uses this parameter along with the numTypes and typeList parameters to determine which files appear in the dialog box.

If this parameter is not nil, CustomGetFilePreview calls the function for each file to determine whether to display the file to the user. The CustomGetFilePreview function supplies you with information identifying the file (see *Inside Macintosh: Files* for more information about the format of this parameter data). Your function returns a Boolean value indicating whether to display the file. Set the Boolean value to false to cause the file to be displayed.

Your function must provide the following interface:

```
pascal Boolean MyFileFilter (ParmBlkPtr parmBlock);
```

numTypes	Specifies the number of file types in the array specified by the typeList parameter (a number between 1 and 4). Set this parameter to -1 to display all files.
typeList	Specifies an array of file types to be displayed to the user. The CustomGetFilePreview function only displays files whose type matches an entry in this array (unless you set the numTypes parameter

## Movie Toolbox

to -1; in this case, the function displays all files to the user). The `SFTypeList` data type is defined as follows:

```
typedef OSType SFTypeList[4];
```

reply	Contains a pointer to a reply structure that is to receive information about the user's selection. See <i>Inside Macintosh: Files</i> for more information about reply structures.
dlgID	Specifies the resource ID of your custom dialog template. You can use this parameter to specify a custom dialog template resource that has a resource type that differs from the standard value. Set this parameter to 0 to use the standard template.
where	Specifies the location of the upper-left corner of the dialog box in global coordinates. If you set this point to (-1, -1), the Movie Toolbox centers the dialog box on the main screen. If you set this point to (-2, -2), the Movie Toolbox centers the dialog box on the screen that has the best display characteristics.
dlgHook	Points to a custom dialog function. You can use this parameter to support a custom dialog box function you have supplied by specifying a dialog template resource in your resource file. You specify the dialog template's resource ID with the <code>dlgID</code> parameter. If you are not supplying a custom dialog function, set this parameter to <code>nil</code> . For more information about using custom dialog functions with the <code>CustomGetFile</code> routine, see <i>Inside Macintosh: Files</i> . For details on the parameters of the custom dialog box function, see "Custom Dialog Functions" on page 2-360.

Your dialog hook function must present the following interface:

```
pascal short MyDlgHook (short item, DialogPtr
                        theDialog, Ptr myDataPtr);
```

## filterProc

Points to your modal-dialog filter function. This function gives you greater control over the interface presented to the user. See *Inside Macintosh: Files* for more information about using modal-dialog filter functions with `CustomGetFile`.

Your modal-dialog filter function must present the following interface.

```
pascal Boolean MyModalFilter (DialogPtr theDialog,
                             EventRecord* theEvent,
                             short itemHit,
                             Ptr myDataPtr);
```

For details on the application-defined modal-dialog filter, see "Modal-Dialog Filter Functions" beginning on page 2-362.



## Movie Toolbox

`activeList`

Contains a pointer to a list of all items in the dialog box that can be activated—that is, made the target of keyboard input. The list is stored as an array of integers. The first integer must contain the number of items in the array (not including this count value). The remaining array entries must contain item numbers that specify valid targets of keyboard input, in the order in which the items are to be activated. Set this parameter to `nil` to direct all keyboard input to the displayed list of filenames.

`activateProc`

Points to your activation function, which controls the highlighting of any items whose shape is known only by your application. See *Inside Macintosh: Files* for more information about standard file activation functions.

Your function must present the following interface:

```
pascal void MyActivateProc (DialogPtr theDialog,
                           short itemNo,
                           Boolean activating,
                           Ptr myDataPtr);
```

`yourDataPtr`

Contains a pointer to optional data that is supplied by your application to your callback functions. When the `CustomGetFilePreview` function calls any of your callback functions, it places this data on the stack, making it available to your functions. Set this parameter to `nil` if you are not supplying any optional data.

## DESCRIPTION

The `CustomGetFilePreview` function is available only if the value of the Gestalt selector `gestaltStandardFileAttr` is true. (See *Inside Macintosh: Overview* for more information about this selector.) This function corresponds to the File Manager's `CustomGetFile` routine. The `CustomGetFilePreview` function takes the same parameters as its existing counterpart with the addition of a `where` parameter that allows you to specify the location of the dialog box. See *Inside Macintosh: Files* for a complete description of the `CustomGetFile` routine and for more information about the parameters to this function.

The `CustomGetFilePreview` function automatically converts files to movies if your application requests movies. If a file could be converted into a movie file using a movie import component, then the file is shown in the Standard File dialog box. See Figure 2-41 on page 2-304 for the dialog box with an automatic file-to-movie conversion option and Figure 2-42 on page 2-305 for the dialog box for saving a movie converted from a file.

**Note**

The `CustomGetFilePreview` function does not appear in the MPW interface file `Movies.h`; rather, it's listed in `ImageCompression.h`. ♦

## Time Base Functions

---

The Movie Toolbox provides a number of functions that allow you to work with time bases. A QuickTime time base defines the time coordinate system of a movie. However, you can also use QuickTime time bases to provide general timing services. This section describes the functions that allow your application to work with time bases. For a complete description of QuickTime time bases, see “Introduction to Movies” beginning on page 2-5.

This section has been divided into the following topics:

- “Creating and Disposing of Time Bases” describes how to create and dispose of time bases and how to assign a time base to a movie
- “Working With Time Base Values” discusses functions that allow your application to work with the contents of a time base
- “Working With Times” describes a number of functions that allow you to convert times between time bases and to perform simple arithmetic on time values
- “Time Base Callback Functions” describes the functions your application may use to condition a time base to invoke functions your application provides

### Note

Time base functions do not change the value of the Movie Toolbox sticky error value. ♦

## Creating and Disposing of Time Bases

---

This section discusses the Movie Toolbox functions your application can use to create and dispose of time bases.

The `NewTimeBase` function lets you create a new time base. You can use the `DisposeTimeBase` function to dispose of a time base once you are finished with it.

Time bases rely on either a clock component or another time base for their time source. You can use the `SetTimeBaseMasterTimeBase` function to cause one time base to be based on another time base. The `GetTimeBaseMasterTimeBase` allows you to determine the master time base of a given time base.

You can assign a clock component to a time base; that clock then acts as the master clock for the time base. You can use the `SetTimeBaseMasterClock` function to assign a clock component to a time base. The `GetTimeBaseMasterClock` function enables you to determine the clock component that is assigned to a time base. You can change the offset between a time base and its time source by calling the `SetTimeBaseZero` function.

You can set the time source of a movie by calling the `SetMovieMasterTimeBase` and `SetMovieMasterClock` functions.

**Note**

Although most time base functions can be used at interrupt time, several of the Movie Toolbox functions cannot. These functions are noted in the sections that follow. ♦

## NewTimeBase

---

The `NewTimeBase` function allows your application to obtain a new time base. This function returns a reference to the new time base. Your application must use that reference with other time base functions.

```
pascal TimeBase NewTimeBase (void);
```

**DESCRIPTION**

The `NewTimeBase` function returns a reference to the new time base.

This function sets the rate of the time base to 0, the start time to its minimum value, the time value to 0, and the stop time to its maximum value.

This function assigns the default clock component to the new time base. If you want to assign a different clock component or a master time base to the new time base, use the `SetTimeBaseMasterClock` or `SetTimeBaseMasterTimeBase` functions, which are described on page 2-318 and page 2-320, respectively.

**SPECIAL CONSIDERATIONS**

The `NewTimeBase` function uses the Memory Manager, so your application must not call it at interrupt time.

**ERROR CODES**

None

## DisposeTimeBase

---

The `DisposeTimeBase` function allows your application to dispose of a time base once you are finished with it.

```
pascal void DisposeTimeBase (TimeBase tb);
```

<code>tb</code>	Specifies the time base for this operation. Your application obtains this time base identifier from the <code>NewTimeBase</code> function described in the previous section.
-----------------	--

**DESCRIPTION**

The `DisposeTimeBase` function cancels and disposes of any pending callback events that are associated with the time base.

**SPECIAL CONSIDERATIONS**

Note that the `DisposeTimeBase` function uses the Memory Manager; therefore, you should not call this function at interrupt time.

**ERROR CODES**

None

## SetMovieMasterClock

---

You can use the `SetMovieMasterClock` function to assign a clock component to a movie. Do not use the `SetTimeBaseMasterClock` function to assign a clock component to a movie.

```
pascal void SetMovieMasterClock (Movie theMovie,
                                Component clockMeister,
                                const TimeRecord *slaveZero);
```

**theMovie** Specifies the movie for this operation. Your application obtains this movie identifier from such functions as `NewMovie`, `NewMovieFromFile`, and `NewMovieFromHandle` (described on page 2-92, page 2-88, and page 2-90, respectively).

**clockMeister** Specifies the clock component to be assigned to this movie. Your application can obtain this component identifier from the Component Manager's `FindNextComponent` routine (see the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for more information about this routine).

**slaveZero** Contains a pointer to the time, in the clock's time scale, that corresponds to a 0 time value for the movie. This parameter allows you to set an offset between the clock component and the time base of the movie. Set this parameter to `nil` if there is no offset.

**ERROR CODES**

None

## SetMovieMasterTimeBase

---

You can use the `SetMovieMasterTimeBase` function to assign a master time base to a movie. Do not use the `SetTimeBaseMasterTimeBase` function (described on page 2-320) to assign a time base to a movie.

```
pascal void SetMovieMasterTimeBase (Movie theMovie, TimeBase tb,
                                     const TimeRecord *slaveZero);
```

<code>theMovie</code>	Specifies the movie for this operation. Your application obtains this movie identifier from such functions as <code>NewMovie</code> , <code>NewMovieFromFile</code> , and <code>NewMovieFromHandle</code> (described on page 2-92, page 2-88, and page 2-90, respectively).
<code>tb</code>	Specifies the master time base to be assigned to this movie. Your application obtains this time base identifier from the <code>NewTimeBase</code> function (described on page 2-316).
<code>slaveZero</code>	Contains a pointer to the time, in the time scale of the master time base, that corresponds to a 0 time value for the movie. This parameter allows you to set an offset between the movie and the master time base. Set this parameter to <code>nil</code> if there is no offset.

### SPECIAL CONSIDERATIONS

The `SetMovieMasterTimeBase` function cannot be called at interrupt time.

### ERROR CODES

None

## SetTimeBaseMasterClock

---

You can use the `SetTimeBaseMasterClock` function to assign a clock component to a time base. A time base derives its time from either a clock component or from another time base. Do not use this function to assign a clock to a movie's time base.

```
pascal void SetTimeBaseMasterClock (TimeBase slave,
                                     Component clockMeister,
                                     const TimeRecord *slaveZero);
```

<code>slave</code>	Specifies the time base for this operation. Your application obtains this time base identifier from the <code>NewTimeBase</code> function (described on page 2-316).
--------------------	--

## Movie Toolbox

`clockMeister`

Specifies the clock component to be assigned to this time base. Your application can obtain this component identifier from the Component Manager's `FindNextComponent` routine (see the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox* for more information about this routine).

`slaveZero`

Contains a pointer to the time, in the clock's time scale, that corresponds to a 0 time value for the slave time base. This parameter allows you to set an offset between the time base and the clock component. Set this parameter to `nil` if there is no offset.

## SPECIAL CONSIDERATIONS

The `SetTimeBaseMasterClock` function cannot be called at interrupt time.

## ERROR CODES

`invalidMovie`    -2010    This movie is corrupted or invalid

## SEE ALSO

You can use the `GetTimeBaseMasterClock` function, which is described in the next section, to determine the clock component that is assigned to a time base.

## GetTimeBaseMasterClock

---

You can use the `GetTimeBaseMasterClock` function to determine the clock component that is assigned to a time base. A time base derives its time from either a clock component or from another time base. If a time base derives its time from a clock component, you can use this function to obtain the component instance of the clock component.

```
pascal ComponentInstance GetTimeBaseMasterClock (TimeBase tb);
```

`tb`

Specifies the time base for this operation. Your application obtains this time base identifier from the `NewTimeBase` function (described on page 2-316).

## DESCRIPTION

The `GetTimeBaseMasterClock` function returns a reference to a component instance of the clock component that provides a time source to this time base.

**Note**

The Component Manager allows a single component to serve multiple client applications at the same time. Each client application has a unique access path to the component. These access paths are called **connections**. You identify a component connection by specifying a **component instance**. The Component Manager provides this component instance to your application when you open a connection to a component. The component maintains separate status information for each open connection. ♦

Do not close this connection—the time base is using the connection to maintain its time source. If a clock component is not assigned to the time base, this function sets the returned reference to `nil`. In this case, the time base relies on another time base for its time source. Use the `GetTimeBaseMasterTimeBase` function, which is described on page 2-321, to obtain the time base reference to that master time base.

**ERROR CODES**

None

**SEE ALSO**

You can use the `SetTimeBaseMasterClock` function, which is described on page 2-318, to assign a clock component to a time base.

## SetTimeBaseMasterTimeBase

---

You can use the `SetTimeBaseMasterTimeBase` function to assign a master time base to a time base. A time base derives its time from either a clock component or another time base. Do not use this function to assign a master time base to a movie's time base.

```
pascal void SetTimeBaseMasterTimeBase (TimeBase slave,
                                         TimeBase master,
                                         const TimeRecord *slaveZero);
```

<code>slave</code>	Specifies the time base for this operation. Your application obtains this time base identifier from the <code>NewTimeBase</code> function (described on page 2-316).
<code>master</code>	Specifies the master time base to be assigned to this time base. Your application obtains this time base identifier from the <code>NewTimeBase</code> function.

## Movie Toolbox

**slaveZero** Contains a pointer to the time, in the time scale of the master time base, that corresponds to a 0 time value for the slave time scale. This parameter allows you to set an offset between the time base and the master time base. Set this parameter to `nil` if there is no offset.

## ERROR CODES

None

## SEE ALSO

You can use the `GetTimeBaseMasterTimeBase` function, which is described in the next section, to determine the master time base that is assigned to a time base.

## GetTimeBaseMasterTimeBase

---

You can use the `GetTimeBaseMasterTimeBase` function to determine the master time base that is assigned to a time base. A time base derives its time from either a clock component or from another time base. If a time base derives its time from another time base, you can use this function to obtain the identifier for that master time base.

```
pascal TimeBase GetTimeBaseMasterTimeBase (TimeBase tb);
```

**tb** Specifies the time base for this operation. Your application obtains this time base identifier from the `NewTimeBase` function (described on page 2-316).

## DESCRIPTION

The `GetTimeBaseMasterTimeBase` function returns a reference to the master time base that provides a time source to this time base. If a master time base is not assigned to the time base, this function sets the returned reference to `nil`. In this case, the time base relies on a clock component for its time source. Use the `GetTimeBaseMasterClock` function, which is described on page 2-319, to obtain the component instance reference to that clock component.

## ERROR CODES

None

## SEE ALSO

You can use the `SetTimeBaseMasterTimeBase` function, which is described in the previous section, to assign a master time base to a time base.



## SetTimeBaseZero

---

You can use the `SetTimeBaseZero` function to change the offset from a time base to either its master time base or its clock component. You establish the initial offset when you assign the time base to its time source.

```
pascal void SetTimeBaseZero (TimeBase tb, TimeRecord *zero);
```

<code>tb</code>	Specifies the time base for this operation. Your application obtains this time base identifier from the <code>NewTimeBase</code> function (described on page 2-316).
<code>zero</code>	Contains a pointer to the time that corresponds to a 0 time value for the slave time scale. This parameter allows you to set an offset between the time base and its time source. Set this parameter to <code>nil</code> if there is no offset.

### ERROR CODES

None

### SEE ALSO

You can use the `SetTimeBaseMasterClock` function (described on page 2-318) to assign a time base to a clock component.

You can use the `SetTimeBaseMasterTimeBase` function (described on page 2-320) to assign a time base to a master time base.

## Working With Time Base Values

---

Every time base contains a rate, a start time, a stop time, a current time, and some status information. The Movie Toolbox provides a number of functions that allow your application to work with the contents of a time base. This section describes those functions.

The `GetTimeBaseTime` function lets you retrieve the current time value of a time base. You can set the current time value by calling the `SetTimeBaseTime` function—this function requires you to provide a time structure. Alternatively, you can set the current time based on a time value by calling the `SetTimeBaseValue` function.

You can determine the rate of a time base by calling the `GetTimeBaseRate` function. You can set the rate of a time base by calling the `SetTimeBaseRate` function. You can determine the effective rate of a specified time base (relative to the master time base to which it is subordinate) by calling the `GetTimeBaseEffectiveRate` function.

## Movie Toolbox

You can retrieve the start time of a time base by calling the `GetTimeBaseStartTime` function. You can set the start time of a time base by calling the `SetTimeBaseStartTime` function. Similarly, you can use the `GetTimeBaseStopTime` and `SetTimeBaseStopTime` functions to work with the stop time of a time base.

The Movie Toolbox provides functions that allow you to work with the status information of a time base. The `GetTimeBaseStatus` function allows you to read the current status of a time base. The `GetTimeBaseFlags` function helps you obtain the control flags of a time base. You can set these flags by calling the `SetTimeBaseFlags` function.

## SetTimeBaseTime

---

The `SetTimeBaseTime` function allows your application to set the current time of a time base. You must specify the new time in a time structure.

```
pascal void SetTimeBaseTime (TimeBase tb, const TimeRecord *tr);
```

<code>tb</code>	Specifies the time base for this operation. Your application obtains this time base identifier from the <code>NewTimeBase</code> function (described on page 2-316).
<code>tr</code>	Contains a pointer to a time structure that contains the current time value.

### DESCRIPTION

If you set the current time of a time base that is the master time base for other time bases, the current times in all the dependent time bases are changed appropriately. If you change the current time in a time base that relies on a master time base, the Movie Toolbox changes the offset between the time base and the master time base—the master time base is not affected.

### ERROR CODES

None

### SEE ALSO

You can set the current time of a time base from a time value by calling the `SetTimeBaseValue` function, which is described in the next section.

## SetTimeBaseValue

---

The `SetTimeBaseValue` function allows your application to set the current time of a time base. You must specify the new time as a time value.

```
pascal void SetTimeBaseValue (TimeBase tb, TimeValue t,
                               TimeScale s);
```

tb	Specifies the time base for this operation. Your application obtains this time base identifier from the <code>NewTimeBase</code> function (described on page 2-316).
t	Specifies the new time value.
s	Specifies the time scale of the new time value.

### DESCRIPTION

If you set the current time of a time base that is the master time base for other time bases, the current times in all the dependent time bases are changed appropriately. If you change the current time in a time base that relies on a master time base, the Movie Toolbox changes the offset between the time base and the master time base—the master time base is not affected.

### ERROR CODES

None

### SEE ALSO

You can set the current time of a time base from a time structure by calling the `SetTimeBaseTime` function, which is described in the previous section.

## GetTimeBaseTime

---

Your application can use the `GetTimeBaseTime` function to obtain the current time value from a time base. You can specify the time scale in which to return the time value.

```
pascal TimeValue GetTimeBaseTime (TimeBase tb, TimeScale s,
                                   TimeRecord *tr);
```

tb	Specifies the time base for this operation. Your application obtains this time base identifier from the <code>NewTimeBase</code> function (described on page 2-316).
----	--

## Movie Toolbox

<code>s</code>	Specifies the time scale in which to return the current time value. Set this parameter to 0 to retrieve the time in the preferred time scale of the time base.
<code>tr</code>	Contains a pointer to a time structure that is to receive the current time value. This is an optional parameter. If you do not want the time value represented in a time structure, set this parameter to <code>nil</code> .

## DESCRIPTION

The `GetTimeBaseTime` function returns a time value that contains the current time from the specified time base in the specified time scale. The function returns this value even if you specify a time structure with the `tr` parameter.

## ERROR CODES

None

## SEE ALSO

You can set the current time of a time base by calling either the `SetTimeBaseTime` or `SetTimeBaseValue` functions, which are described on page 2-323 and page 2-324, respectively.

## SetTimeBaseRate

---

The `SetTimeBaseRate` function allows your application to set the rate of a time base.

```
pascal void SetTimeBaseRate (TimeBase tb, Fixed r);
```

<code>tb</code>	Specifies the time base for this operation. Your application obtains this time base identifier from the <code>NewTimeBase</code> function (described on page 2-316).
<code>r</code>	Specifies the rate of the time base.

## DESCRIPTION

You can determine the number of time units that pass each second for a time base by multiplying its rate by the time scale of its time coordinate system. For example, if you set the rate of a time base to 2 and the time base has a time scale of 2, that time base passes through 4 units of its time each second.

Rates may be set to negative values. Negative rates cause time to move backward for the time base.

**ERROR CODES**

None

**SEE ALSO**

You can retrieve the rate of a time base by calling the `GetTimeBaseRate` function, which is described in the next section.

**GetTimeBaseRate**

---

The `GetTimeBaseRate` function allows your application to retrieve the rate of a time base.

Rates may be set to negative values. Negative rates cause time to move backward for the time base.

```
pascal Fixed GetTimeBaseRate (TimeBase tb);
```

**tb** Specifies the time base for this operation. Your application obtains this time base identifier from the `NewTimeBase` function (described on page 2-316).

**DESCRIPTION**

The `GetTimeBaseRate` function returns the current rate of the time base as a fixed-point number. This rate value may be nonzero even if the time base has stopped, because it has reached its stop time.

**ERROR CODES**

None

**GetTimeBaseEffectiveRate**

---

The `GetTimeBaseEffectiveRate` function returns the effective rate at which the specified time base is moving, relative to its master clock.

```
pascal Fixed GetTimeBaseEffectiveRate (TimeBase tb);
```

## Movie Toolbox

**tb** Specifies the time base for this operation. Your application obtains this time base identifier from the `NewTimeBase` function (described on page 2-316).

**DESCRIPTION**

The `GetTimeBaseEffectiveRate` function is useful when you need to make scheduling decisions based on the rate of a time base—for example, when you are writing a media handler. (For more on media handlers, see *Inside Macintosh: QuickTime Components*.) By calling `GetTimeBaseEffectiveRate` rather than the `GetTimeBaseRate` function (described in the previous section), you can easily take into account any time base subordination that may be in effect.

**SetTimeBaseStartTime**

---

You can set the start time of a time base by calling the `SetTimeBaseStartTime` function. The start time defines the time base's minimum time value. You must specify the new start time in a time structure.

```
pascal void SetTimeBaseStartTime (TimeBase tb,
                                   const TimeRecord *tr);
```

**tb** Specifies the time base for this operation. Your application obtains this time base identifier from the `NewTimeBase` function (described on page 2-316).

**tr** Contains a pointer to a time structure that contains the start time value.

**DESCRIPTION**

Do not use this function to restrict the Movie Toolbox to a portion of a movie—use the `SetMovieActiveSegment` function (described on page 2-136) instead.

**ERROR CODES**

None

**SEE ALSO**

You can determine the start time of a time base by calling the `GetTimeBaseStartTime` function, which is described in the next section.

## GetTimeBaseStartTime

---

You can determine the start time of a time base by calling the `GetTimeBaseStartTime` function.

```
pascal TimeValue GetTimeBaseStartTime (TimeBase tb, TimeScale s,
                                       TimeRecord *tr);
```

<code>tb</code>	Specifies the time base for this operation. Your application obtains this time base identifier from the <code>NewTimeBase</code> function (described on page 2-316).
<code>s</code>	Specifies the time scale in which to return the start time.
<code>tr</code>	Contains a pointer to a time structure that is to receive the start time. This is an optional parameter. If you do not want the time value represented in a time structure, set this parameter to <code>nil</code> .

### DESCRIPTION

The `GetTimeBaseStartTime` returns a time value that contains the start time from the specified time base in the specified time scale. The function returns this value even if you specify a time structure with the `tr` parameter.

### ERROR CODES

None

### SEE ALSO

You can set the start time of a time base by calling the `SetTimeBaseStartTime` function, which is described in the previous section.

## SetTimeBaseStopTime

---

You can set the stop time of a time base by calling the `SetTimeBaseStopTime` function. The stop time defines the time base's maximum time value. You must specify the new stop time in a time structure.

```
pascal void SetTimeBaseStopTime (TimeBase tb,
                                  const TimeRecord *tr);
```

<code>tb</code>	Specifies the time base for this operation. Your application obtains this time base identifier from the <code>NewTimeBase</code> function (described on page 2-316).
<code>tr</code>	Contains a pointer to a time structure that contains the stop time value.

**DESCRIPTION**

Do not use the `SetTimeBaseStopTime` function to restrict the Movie Toolbox to a portion of a movie—use the `SetMovieActiveSegment` function (described on page 2-136) instead.

**ERROR CODES**

None

**SEE ALSO**

You can determine the stop time of a time base by calling the `GetTimeBaseStopTime` function, which is described in the next section.

## GetTimeBaseStopTime

---

You can determine the stop time of a time base by calling the `GetTimeBaseStopTime` function.

```
pascal TimeValue GetTimeBaseStopTime (TimeBase tb, TimeScale s,  
                                       TimeRecord *tr);
```

<code>tb</code>	Specifies the time base for this operation. Your application obtains this time base identifier from the <code>NewTimeBase</code> function (described on page 2-316).
<code>s</code>	Specifies the time scale in which to return the stop time.
<code>tr</code>	Contains a pointer to a time structure that is to receive the stop time. This is an optional parameter. If you do not want the time value represented in a time structure, set this parameter to <code>nil</code> .

**DESCRIPTION**

The `GetTimeBaseStopTime` returns a time value that contains the stop time from the specified time base in the specified time scale. The function returns this value even if you specify a time structure with the `out` parameter.

**ERROR CODES**

None

**SEE ALSO**

You can set the stop time of a time base by calling the `SetTimeBaseStopTime` function, which is described in the previous section.



## SetTimeBaseFlags

---

The `SetTimeBaseFlags` function allows your application to set the contents of the control flags of a time base.

```
pascal void SetTimeBaseFlags (TimeBase tb, long timeBaseFlags);
```

**tb** Specifies the time base for this operation. Your application obtains this time base identifier from the `NewTimeBase` function (described on page 2-316).

**timeBaseFlags** Specifies the control flags for this time base. The following flags are defined. You may set only one flag to 1 (be sure to set unused flags to 0):

**loopTimeBase**

Indicates whether the time base loops. If you set this flag to 1 and the rate is positive, the time base loops back and restarts from its start time when it reaches its stop time. If you set this flag to 1 and the rate is negative, the time base loops to its stop time. If you set the flag to 0, the movie stops when it reaches the end.

**palindromeLoopTimeBase**

Indicates whether the time base loops in a palindrome fashion. **Palindrome looping** causes a time base to move alternately forward and backward. Set this flag to 1 to cause the time base to loop in this manner.

### ERROR CODES

None

### SEE ALSO

You can retrieve the control flags of a time base by calling the `GetTimeBaseFlags` function, which is described in the next section.

## GetTimeBaseFlags

---

The `GetTimeBaseFlags` function allows your application to obtain the contents of the control flags of a time base.

```
pascal long GetTimeBaseFlags (TimeBase tb);
```

**tb** Specifies the time base for this operation. Your application obtains this time base identifier from the `NewTimeBase` function (described on page 2-316).

**DESCRIPTION**

The `GetTimeBaseFlags` function returns the control flags of a time base. The following flags are defined (unused flags are set to 0):

**loopTimeBase**

Indicates whether the time base loops. If this flag is set to 1 and the rate is positive, the time base loops back and restarts from its start time when it reaches its stop time. If this flag is set to 1 and the rate is negative, the time base loops to its stop time. If the flag is set to 0, the movie stops when it reaches the end.

**palindromeLoopTimeBase**

Indicates whether the time base loops in a palindrome fashion. Palindrome looping causes a time base to move alternately forward and backward. If this flag is set to 1, the time base is palindrome looping.

**ERROR CODES**

None

**SEE ALSO**

You can set the control flags of a time base by calling the `SetTimeBaseFlags` function, which is described in the previous section.

**GetTimeBaseStatus**

---

Your application can retrieve status information from a time base by calling the `GetTimeBaseStatus` function. This status information allows you to determine when the current time of a time base would fall outside of the range of values specified by the start and stop times of the time base. This can happen when a time base relies on a master time base or when its time has reached the stop time.

```
pascal long GetTimeBaseStatus (TimeBase tb,
                                TimeRecord *unpinnedTime);
```

**tb** Specifies the time base for this operation. Your application obtains this time base identifier from the `NewTimeBase` function (described on page 2-316).

**unpinnedTime**

Contains a pointer to a time structure that is to receive the current time of the time base. Note that this time value may be outside the range of values specified by the start and stop times of the time base.

## Movie Toolbox

## DESCRIPTION

The `GetTimeBaseStatus` function returns flags that indicate whether the returned time value is outside the range of values specified by the start and stop times of the time base. The following flags are defined (unused flags are set to 0):

`timeBaseBeforeStartTime`

Indicates that the time value represented by the contents of the time structure referred to by the `unpinnedTime` parameter lies before the start time of the time base. The Movie Toolbox sets this flag to 1 if the current time is before the start time of the time base.

`timeBaseAfterStopTime`

Indicates that the time value represented by the contents of the time structure referred to by the `unpinnedTime` parameter lies after the stop time of the time base. The Movie Toolbox sets this flag to 1 if the current time is after the stop time of the time base.

## ERROR CODES

None

## Working With Times

The Movie Toolbox provides a number of functions that allow you to work with time structures. This section describes those functions.

All of these functions work with time structures (see “The Time Structure” on page 2-77 for a complete discussion of the time structure). You can use time structures to represent either time values or durations. Time values specify a point in time, relative to a given time base. Durations specify a span of time, relative to a given time scale. Durations are represented by time structures that have the time base set to 0 (that is, the `base` field in the time structure is set to `nil`).

You can use the `ConvertTime` function to convert a time you obtain from one time base into a time that is relative to another time base. Similarly, you can use the `ConvertTimeScale` function to convert a time from one time scale to another.

You can add two times by calling the `AddTime` function; you can subtract two times with the `SubtractTime` function.

## AddTime

The `AddTime` function adds two times. You must specify the times in time structures.

```
pascal void AddTime (TimeRecord *dst, const TimeRecord *src);
```

`dst`                      Contains a pointer to a time structure. This time structure contains one of the operands for the addition. The `AddTime` function returns the result of the addition into this time structure.

Movie Toolbox

**src** Contains a pointer to a time structure. The Movie Toolbox adds this value to the time or duration specified by the **dst** parameter.

DESCRIPTION

If these times are relative to different time scales or time bases, the `AddTime` function converts the times as appropriate to yield reasonable results. However, the time bases for both time values must rely on the same time source.

The result value is formatted based on the operands as follows:

<b>dst</b>	<b>src</b>	<b>Result</b>
Duration	Duration	Duration
Time value	Duration	Time value

ERROR CODES

None

SubtractTime

---

The `SubtractTime` function subtracts one time from another. You must specify the times in time structures.

```
pascal void SubtractTime (TimeRecord *dst, const TimeRecord *src);
```

**dst** Contains a pointer to a time structure. This time structure contains one of the operands for the subtraction. The `SubtractTime` function returns the result of the subtraction into this time structure.

**src** Contains a pointer to a time structure. The Movie Toolbox subtracts this value from the time or duration specified by the **dst** parameter.

DESCRIPTION

If these times are relative to different time scales or time bases, the `SubtractTime` function converts the times as appropriate to yield reasonable results. However, the time bases for both time values must rely on the same time source.

The result value is formatted based on the operands as follows:

<b>dst</b>	<b>src</b>	<b>Result</b>
Time value	Duration	Duration
Duration	Duration	Duration
Time value	Time value	Duration

## ERROR CODES

None

**ConvertTime**

---

You can convert a time you obtain from one time base into a time that is relative to another time base by calling the `ConvertTime` function. Both time bases must rely on the same time source. You must specify the time to be converted in a time structure.

```
pascal void ConvertTime (TimeRecord *inout, TimeBase newBase);
```

<code>inout</code>	Contains a pointer to a time structure that contains the time value to be converted. The <code>ConvertTime</code> function replaces the contents of this time structure with the time value relative to the specified time base.
<code>newBase</code>	Specifies the time base for this operation. Your application obtains this time base identifier from the <code>NewTimeBase</code> function (described on page 2-316).

## DESCRIPTION

The `ConvertTime` function includes the rate associated with each time value in the conversion; therefore, you should use this function when you want to convert time values. Use the `ConvertTimeScale` function (described in the next section) to convert durations.

## ERROR CODES

None

**ConvertTimeScale**

---

You can convert a time from one time scale into a time that is relative to another time base by calling the `ConvertTimeScale` function. You must specify the time to be converted in a time structure.

```
pascal void ConvertTimeScale (TimeRecord *inout,
                               TimeScale newScale);
```

<code>inout</code>	Contains a pointer to a time structure that contains the time value to be converted. The <code>ConvertTimeScale</code> function replaces the contents of this time structure with the time value relative to the specified time scale.
<code>newScale</code>	Specifies the time scale for this operation.

**DESCRIPTION**

The `ConvertTimeScale` function does not include the rate associated with the time value in the conversion; therefore, you should use this function when you want to convert time durations, but not when converting time values. Use the `ConvertTime` function (described in the previous section) to convert time values.

**ERROR CODES**

None

## Time Base Callback Functions

---

If your application uses QuickTime time bases, it may define callback functions that are associated with a specific time base. Your application can then use these callback functions to perform activities that are triggered by temporal events, such as a certain time being reached or a specified rate being achieved. The time base functions of the Movie Toolbox interact with clock components to schedule the invocation of these callback functions—clock components are responsible for invoking the callback function at its scheduled time. Your application can use the functions described in this section to establish your own callback function and to schedule callback events.

You can define three types of callback events. These types are distinguished by the nature of the temporal event that triggers the Movie Toolbox to call your function. The three types are

- events that are triggered at a specified time
- events that are triggered when the rate reaches a specified value
- events that are triggered when the time value of a time base changes by an amount different from the time base's rate

You specify a callback event's type when you define the callback event, using the `NewCallBack` function.

You specify whether your event can occur at interrupt time when you define the callback event, using the `NewCallBack` function. Your function is called closer to the triggering event at interrupt time, but it is subject to all the restrictions of interrupt functions (for example, your callback function cannot cause memory to be moved). If your function is not called at interrupt time, you are free of these restrictions—but your function may be called later, because the invocation is delayed to avoid interrupt time.

The `NewCallBack` function allocates the memory to support a callback event. When you are done with the callback event, you dispose of it by calling the `DisposeCallBack` function.

You schedule a callback event by calling the `CallMeWhen` function. Call `CancelCallBack` function to unschedule a callback event.

You can retrieve the time base of a callback event by calling the `GetCallBackTimeBase` function. You can obtain the type of a callback event by calling the `GetCallBackType` function.

## NewCallback

---

The `NewCallback` function creates a new callback event. The callback event created at this time is not active until you schedule it by calling the `CallMeWhen` function, which is described in the next section.

### ▲ WARNING

You must not call this function at interrupt time. ▲

```
pascal QTCallback NewCallback (TimeBase tb, short cbType);
```

**tb** Specifies the callback event's time base. You obtain this identifier from the `NewTimeBase` function (described on page 2-316).

**cbType** Specifies when the callback event is to be invoked. The value of this field governs how the Movie Toolbox interprets the data supplied in the `param1`, `param2`, and `param3` parameters to the `CallMeWhen` function, which is described in the next section. The following values are valid for this parameter:

`callbackAtTime`

Indicates that the event is to be invoked at a specified time.

`callbackAtRate`

Indicates that the event is to be invoked when the rate for the time base reaches a specified value.

`callbackAtTimeJump`

Indicates that the event is to be invoked when the time base's time value changes by an amount that differs from its rate.

`callbackAtExtremes`

Indicates that the event is to be invoked when the time base reaches its start time or its stop time. If the start or stop time of the time base changes, the call back is automatically rescheduled. This is very useful for looping or determining when a movie is complete. You determine when the callback is to be fired with the `triggerAtStart` and `triggerAtStop` constants. Both flags may be set.

In addition, if the high-order bit of the `cbType` parameter is set to 1 (this bit is defined by the `callbackAtInterrupt` flag), the event can be invoked at interrupt time.

### DESCRIPTION

The `NewCallback` function returns a reference to the new callback event. You must provide this reference to other Movie Toolbox functions described in this section. If the Movie Toolbox cannot create the callback event, this function returns `nil`.

## ERROR CODES

None

**CallMeWhen**

You schedule a callback event by calling the `CallMeWhen` function. You can call this function from your callback function.

```
pascal OSErr CallMeWhen (QTCallback cb,
                        QTCallbackProc callbackProc,
                        long refcon, long param1,
                        long param2, long param3);
```

**cb** Specifies the callback event for the operation. You obtain this identifier from the `NewCallback` function, which is described in the previous section.

**callbackProc**

Points to your callback function.

Your callback function must have the following form:

```
pascal void MyCallbackProc (QTCallback cb,
                           long refcon);
```

See “Callback Event Functions” on page 2-364 for details.

**refcon** Contains a reference constant value for your callback function.

**param1** Contains scheduling information. The Movie Toolbox interprets this parameter based on the value of the `cbType` parameter to the `NewCallback` function, described in the previous section.

If `cbType` is set to `callbackAtTime`, the `param1` parameter contains flags indicating when to invoke your callback function for this callback event. The following values are defined (be sure to set unused flags to 0):

**triggerTimeFwd**

Indicates that your callback function should be called at the time specified by `param2` only when time is moving forward (positive rate). The value of this flag is 0x0001.

**triggerTimeBwd**

Indicates that your callback function should be called at the time specified by `param2` only when time is moving backward (negative rate). The value of this flag is 0x0002.

**triggerTimeEither**

Indicates that your callback function should be called at the time specified by `param2` without regard to direction, but the rate must be nonzero. The value of this flag is 0x0003.



## Movie Toolbox

If the `cbType` parameter is set to `callBackAtRate`, `param1` contains flags indicating when to invoke your callback function for this event. The following values are defined (be sure to set unused flags to 0):

`triggerRateChange`

Indicates that your callback function should be called whenever the rate changes. The value of this flag is 0x0000.

`triggerRateLT`

Indicates that your callback function should be called when the rate changes to a value less than that specified by `param2`. The value of this flag is 0x0004.

`triggerRateGT`

Indicates that your callback function should be called when the rate changes to a value greater than that specified by `param2`. The value of this flag is 0x0008.

`triggerRateEqual`

Indicates that your callback function should be called when the rate changes to a value equal to that specified by `param2`. The value of this flag is 0x0010.

`triggerRateLTE`

Indicates that your callback function should be called when the rate changes to a value that is less than or equal to that specified by `param2`. The value of this flag is 0x0014.

`triggerRateGTE`

Indicates that your callback function should be called when the rate changes to a value that is less than or equal to that specified by `param2`. The value of this flag is 0x0018.

`triggerRateNotEqual`

Indicates that your callback function should be called when the rate changes to a value that is not equal to that specified by `param2`. The value of this flag is 0x001C.

`param2`

Contains scheduling information. The Movie Toolbox interprets this parameter based on the value of the `cbType` parameter to the `NewCallBack` function, described in the previous section.

If `cbType` is set to `callBackAtTime`, the `param2` parameter contains the time value at which your callback function is to be invoked for this event. The `param1` parameter contains flags affecting when the Movie Toolbox calls your function.

If `cbType` is set to `callBackAtRate`, the `param2` parameter contains the rate value at which your callback function is to be invoked for this event. The `param1` parameter contains flags affecting when the Movie Toolbox calls your function.

`param3`

Contains the time scale in which to interpret the time value that is stored in `param3` if `cbType` is set to `callBackAtTime`.

## ERROR CODES

None

## CancelCallback

---

You use the `CancelCallback` function to cancel a callback event before it executes.

```
pascal void CancelCallback (QTCallback cb);
```

**cb** Specifies the callback event for this operation. You obtain this value from the `NewCallback` function (described on page 2-336).

## DESCRIPTION

The `CancelCallback` function removes the callback event from the list of callback events maintained by the Movie Toolbox. The Movie Toolbox calls this function automatically when it invokes your callback function. In order for a callback event to be scheduled, you must call the `CallMeWhen` function, which is described in the previous section.

## ERROR CODES

None

## DisposeCallback

---

The `DisposeCallback` function disposes of the memory associated with the specified callback event and cancels the event if it is pending. You should call this function when you are done with each callback event.

**▲ WARNING**

You must not call this function at interrupt time. ▲

```
pascal void DisposeCallback (QTCallback cb);
```

**cb** Specifies the callback event for the operation. You obtain this value from the `NewCallback` function (described on page 2-336).

## ERROR CODES

None

## GetCallbackTimeBase

---

You can retrieve the time base of a callback event by calling the `GetCallbackTimeBase` function. Your application specifies the callback event's time base by calling the `NewCallback` function, which is described on page 2-336.

```
pascal TimeBase GetCallbackTimeBase (QTCallback cb);
```

**cb** Specifies the callback event for the operation. You obtain this value from the `NewCallback` function.

### DESCRIPTION

The `GetCallbackTimeBase` function returns a reference to the callback event's time base.

### ERROR CODES

None

## GetCallbackType

---

You can retrieve a callback event's type by calling the `GetCallbackType` function. You specify the type value when you call the `NewCallback` function (described on page 2-336).

```
pascal short GetCallbackType (QTCallback cb);
```

**cb** Specifies the callback event for the operation. You obtain this value from the `NewCallback` function.

### DESCRIPTION

The `GetCallbackTimeBase` function returns the callback event's type value. The following values are valid:

`callbackAtTime`

Indicates that the event is to be invoked at a specified time.

`callbackAtRate`

Indicates that the event is to be invoked when the rate for the time base reaches a specified value.

## Movie Toolbox

`callbackAtTimeJump`

Indicates that the event is to be invoked when the time base's time value changes by an amount that differs from its rate.

In addition, if the high-order bit of the returned value is set to 1 (this bit is defined by the `callbackAtInterrupt` flag), the event can be invoked at interrupt time.

## ERROR CODES

None

## Matrix Functions

---

The Movie Toolbox provides a number of functions that allow you to work with transformation matrices. This section describes those functions. For more information about transformation matrices, see “The Transformation Matrix” on page 2-26. For descriptions of fixed-point and fixed-rectangle structures, see “The Fixed-Point and Fixed-Rectangle Structures” on page 2-78.

**Note**

The functions described in this section do not appear in the MPW interface file `Movies.h`; rather, they appear in the `ImageCompression.h` interface file. ♦

## SetIdentityMatrix

---

The `SetIdentityMatrix` function allows your application to set the contents of a matrix so that it performs no transformation. Such matrices are referred to as *identity matrices*.

```
pascal void SetIdentityMatrix (MatrixRecord *matrix);
```

**matrix**      Contains a pointer to a matrix structure. The `SetIdentityMatrix` function updates the contents of this matrix so that the matrix describes the identity matrix.

## ERROR CODES

None

## GetMatrixType

---

The `GetMatrixType` function allows your application to obtain information about a matrix. This information indicates the nature of the transformation defined by the matrix.

```
pascal short GetMatrixType (MatrixRecordPtr m);
```

`m`                      Points to the matrix for this operation.

### DESCRIPTION

The `GetMatrixType` function returns an integer that indicates the nature of the transformation defined by the matrix. The following values are possible:

`identityMatrixType`

Indicates that the specified matrix is an identity matrix.

`translateMatrixType`

Indicates that the specified matrix defines a translation operation.

`scaleMatrixType`

Indicates that the specified matrix defines a scaling operation.

`scaleTranslateMatrixType`

Indicates that the specified matrix defines both a translation operation and a scaling operation.

`linearMatrixType`

Indicates that the specified matrix defines a rotation, skew, or shear operation.

`linearTranslateMatrixType`

Indicates that the specified matrix defines both a translation operation and a rotation, skew, or shear operation.

`perspectiveMatrixType`

Indicates that the specified matrix defines a perspective (nonlinear) operation.

### ERROR CODES

None

## CopyMatrix

---

The `CopyMatrix` function copies the contents of one matrix into another matrix.

```
pascal void CopyMatrix (MatrixRecordPtr m1, MatrixRecord *m2);
```

- |    |   |
|----|---|
| m1 | Specifies the source matrix for the copy operation.   |
| m2 | Contains a pointer to the destination matrix for the copy operation. The <code>CopyMatrix</code> function copies the values from the matrix specified by the m1 parameter into this matrix. |

### DESCRIPTION

The `CopyMatrix` function is a convenience function for copying the contents of one matrix to another. You can achieve the same results by using the Memory Manager's `BlockMove` routine, or by assigning the contents of one matrix record to another directly.

### ERROR CODES

None

## EqualMatrix

---

The `EqualMatrix` function compares two matrices and returns a result that indicates whether the matrices are equal.

```
pascal Boolean EqualMatrix (const MatrixRecord *m1,  
                           const MatrixRecord *m2);
```

- |    |   |
|----|---|
| m1 | Contains a pointer to one matrix for the compare operation.       |
| m2 | Contains a pointer to the other matrix for the compare operation. |

### DESCRIPTION

The `EqualMatrix` function returns a Boolean value that indicates whether the specified matrices are equal. If the matrices are equal, the function sets this returned value to `true`. Otherwise, it sets the returned value to `false`.

### ERROR CODES

None

## TranslateMatrix

---

The `TranslateMatrix` function allows your application to add a translation value to a specified matrix.

```
pascal void TranslateMatrix (MatrixRecord *m,
                             Fixed deltaH, Fixed deltaV);
```

<code>m</code>	Contains a pointer to the matrix structure for this operation.
<code>deltaH</code>	Specifies the value to be added to the x coordinate translation value.
<code>deltaV</code>	Specifies the value to be added to the y coordinate translation value.

### ERROR CODES

None

## ScaleMatrix

---

The `ScaleMatrix` function allows your application to modify the contents of a matrix so that it defines a scaling operation.

```
pascal void ScaleMatrix (MatrixRecord *m, Fixed scaleX,
                         Fixed scaleY, Fixed aboutX, Fixed aboutY);
```

<code>m</code>	Contains a pointer to a matrix structure. The <code>ScaleMatrix</code> function updates the contents of this matrix so that the matrix describes a scaling operation—that is, it concatenates the respective transformations onto whatever was initially in the matrix structure. You specify the magnitude of the scaling operation with the <code>scaleX</code> and <code>scaleY</code> parameters. You specify the anchor point with the <code>aboutX</code> and <code>aboutY</code> parameters.
<code>scaleX</code>	Specifies the scaling factor applied to x coordinates.
<code>scaleY</code>	Specifies the scaling factor applied to y coordinates.
<code>aboutX</code>	Specifies the x coordinate of the anchor point.
<code>aboutY</code>	Specifies the y coordinate of the anchor point.

### ERROR CODES

None

## RotateMatrix

---

The `RotateMatrix` function allows your application to modify the contents of a matrix so that it defines a rotation operation.

```
pascal void RotateMatrix (MatrixRecord *m, Fixed degrees,
                          Fixed aboutX, Fixed aboutY);
```

m	Contains a pointer to a matrix structure. The <code>RotateMatrix</code> function updates the contents of this matrix so that the matrix describes a rotation operation—that is, it concatenates the rotation transformations onto whatever was initially in the matrix structure. You specify the direction and amount of rotation with the <code>degrees</code> parameter. You specify the point of rotation with the <code>aboutX</code> and <code>aboutY</code> parameters.
degrees	Specifies the number of degrees of rotation.
aboutX	Specifies the x coordinate of the anchor point of rotation.
aboutY	Specifies the y coordinate of the anchor point of rotation.

### ERROR CODES

None

## SkewMatrix

---

The `SkewMatrix` function allows your application to modify the contents of a matrix so that it defines a skew transformation. A skew operation alters the display of an element along one dimension—for example, converting a rectangle into a parallelogram is a skew operation.

```
pascal void SkewMatrix (MatrixRecord *m, Fixed skewX, Fixed skewY,
                        Fixed aboutX, Fixed aboutY);
```

m	Contains a pointer to the matrix for this operation. The <code>SkewMatrix</code> function updates the contents of this matrix so that it defines a skew operation—that is, it concatenates the respective transformations onto whatever was initially in the matrix structure. You specify the magnitude and direction of the skew operation with the <code>skewX</code> and <code>skewY</code> parameters. You specify an anchor point with the <code>aboutX</code> and <code>aboutY</code> parameters.
skewX	Specifies the skew value to be applied to x coordinates.



## Movie Toolbox

<code>skewY</code>	Specifies the skew value to be applied to y coordinates.
<code>aboutX</code>	Specifies the x coordinate of the anchor point.
<code>aboutY</code>	Specifies the y coordinate of the anchor point.

## ConcatMatrix

---

The `ConcatMatrix` function concatenates two matrices, combining the transformations described by both matrices into a single matrix.

```
pascal void ConcatMatrix (MatrixRecord *a, MatrixRecord *b);
```

<code>a</code>	Contains a pointer to the source matrix.
<code>b</code>	Contains a pointer to the destination matrix. The <code>ConcatMatrix</code> function performs a matrix multiplication operation, combining the two matrices, and leaves the result in the matrix specified by this parameter.

### DESCRIPTION

The form of the operation that the `ConcatMatrix` function performs is shown by the following formula:

$$[B] = [B] \times [A]$$

This is a matrix multiplication operation. Note that matrix multiplication is not commutative.

### ERROR CODES

None

## InverseMatrix

---

The `InverseMatrix` function creates a new matrix that is the inverse of a specified matrix.

```
pascal Boolean InverseMatrix (MatrixRecord *m,
                             MatrixRecord *im);
```

## Movie Toolbox

<code>m</code>	Contains a pointer to the source matrix for the operation.
<code>im</code>	Contains a pointer to a matrix structure that is to receive the new matrix. The <code>InverseMatrix</code> function updates this structure so that it contains a matrix that is the inverse of that specified by the <code>m</code> parameter.

## DESCRIPTION

The `InverseMatrix` function returns a Boolean value that indicates whether it could create an inverse matrix. If the function could create an inverse matrix, it sets this returned value to `true`. Otherwise, the function sets the returned value to `false`.

## ERROR CODES

None

## TransformPoints

---

The `TransformPoints` function allows your application to transform a set of QuickDraw points through a specified matrix.

```
pascal OSErr TransformPoints (MatrixRecord *mp, Point *pt1,
                              long count);
```

<code>mp</code>	Contains a pointer to the transformation matrix for this operation.
<code>pt1</code>	Contains a pointer to the first QuickDraw point to be transformed.
<code>count</code>	Specifies the number of QuickDraw points to be transformed. These points must be stored immediately following the point specified by the <code>pt1</code> parameter.

## ERROR CODES

None

## SEE ALSO

You can transform a set of QuickDraw points that are made up of fixed values by calling the `TransformFixedPoints` function, which is described in the next section.

## TransformFixedPoints

---

The `TransformFixedPoints` function allows your application to transform a set of fixed points through a specified matrix.

```
pascal OSErr TransformFixedPoints (MatrixRecord *m,
                                   FixedPoint *fpt, long count);
```

<code>m</code>	Contains a pointer to the transformation matrix for this operation.
<code>fpt</code>	Contains a pointer to the first fixed point to be transformed.
<code>count</code>	Specifies the number of fixed points to be transformed. These points must be stored immediately following the point specified by the <code>fpt</code> parameter.

### ERROR CODES

None

### SEE ALSO

You can transform a set of fixed points that is made up of short integer values by calling the `TransformPoints` function, which is described in the previous section.

## TransformRect

---

The `TransformRect` function allows your application to transform the upper-left and lower-right points of a rectangle through a specified matrix.

```
pascal Boolean TransformRect (MatrixRecordPtr m, Rect *r,
                              FixedPoint *fpp);
```

<code>m</code>	Specifies the matrix for this operation.
<code>r</code>	Contains a pointer to the structure that defines the rectangle to be transformed. The <code>TransformRect</code> function returns the updated coordinates into the structure referred to by this parameter. If the resulting rectangle has been rotated or skewed (that is, the transformation involves operations other than scaling and translation), the function sets the returned Boolean value to <code>false</code> and returns the coordinates of the rectangle that encloses the transformed rectangle. The function then updates the points specified by the <code>fpp</code> parameter to contain the coordinates of the four corners of the transformed rectangle.

## Movie Toolbox

`fpp` Contains a pointer to an array of four fixed points. The `TransformRect` function returns the coordinates of the four corners of the rectangle after the transformation operation.

If you do not want this information, set this parameter to `nil`.

## DESCRIPTION

The `TransformRect` function returns a Boolean value indicating the nature of the result rectangle. If the matrix defines transformations other than translation and scaling, the `TransformRect` function sets the returned value to `false`, updates the rectangle specified by the `r` parameter to define the boundary box of the resulting rectangle, and places the coordinates of the corners of the resulting rectangle in the points specified by the `fpp` parameter. If the transformed rectangle and its boundary box are the same, the function sets the returned value to `true`.

## ERROR CODES

None

**TransformFixedRect**

The `TransformFixedRect` function allows your application to transform the upper-left and lower-right points of a rectangle through a specified matrix. This rectangle must be specified by fixed points.

```
pascal Boolean TransformFixedRect (MatrixRecord *m,
                                   FixedRect *fr,
                                   FixedPoint *fpp);
```

`m` Contains a pointer to the matrix for this operation.

`fr` Contains a pointer to the structure that defines the rectangle to be transformed. The `TransformFixedRect` function returns the updated coordinates into the structure referred to by this parameter. If the resulting rectangle has been rotated or skewed (that is, the transformation involves operations other than scaling and translation), the function sets the returned Boolean value to `false` and returns the coordinates of the boundary box of the transformed rectangle. The function then updates the points specified by the `fpp` parameter to contain the coordinates of the four corners of the transformed rectangle.

`fpp` Contains a pointer to an array of four fixed points. The `TransformFixedRect` function returns the coordinates of the four corners of the rectangle after the transformation operation.

If you do not want this information, set this parameter to `nil`.

## Movie Toolbox

## DESCRIPTION

The `TransformFixedRect` function returns a Boolean value indicating the nature of the result rectangle. If the matrix defines transformations other than translation and scaling, the `TransformFixedRect` function sets the returned value to `false`, updates the rectangle specified by the `fr` parameter to define the boundary box of the resulting rectangle, and places the coordinates of the corners of the resulting rectangle in the points specified by the `fpp` parameter. If the transformed rectangle and its boundary box are the same, the function sets the returned value to `true`.

## ERROR CODES

None

## SEE ALSO

You can transform a standard rectangle by calling the `TransformRect` function, which is described in the previous section.

## TransformRgn

---

The `TransformRgn` function allows your application to apply a specified matrix to a region.

```
pascal OSErr TransformRgn (MatrixRecordPtr mp, RgnHandle r);
```

<code>mp</code>	Points to the matrix for this operation. The <code>TransformRgn</code> function currently supports only translating and scaling operations.
<code>r</code>	Specifies the region to be transformed. The <code>TransformRgn</code> function transforms each point in the region according to the contents of the specified matrix.

## ERROR CODES

Memory Manager errors

## RectMatrix

The `RectMatrix` function allows your application to create a matrix that performs a translate and scale operation as described by the relationship between two rectangles.

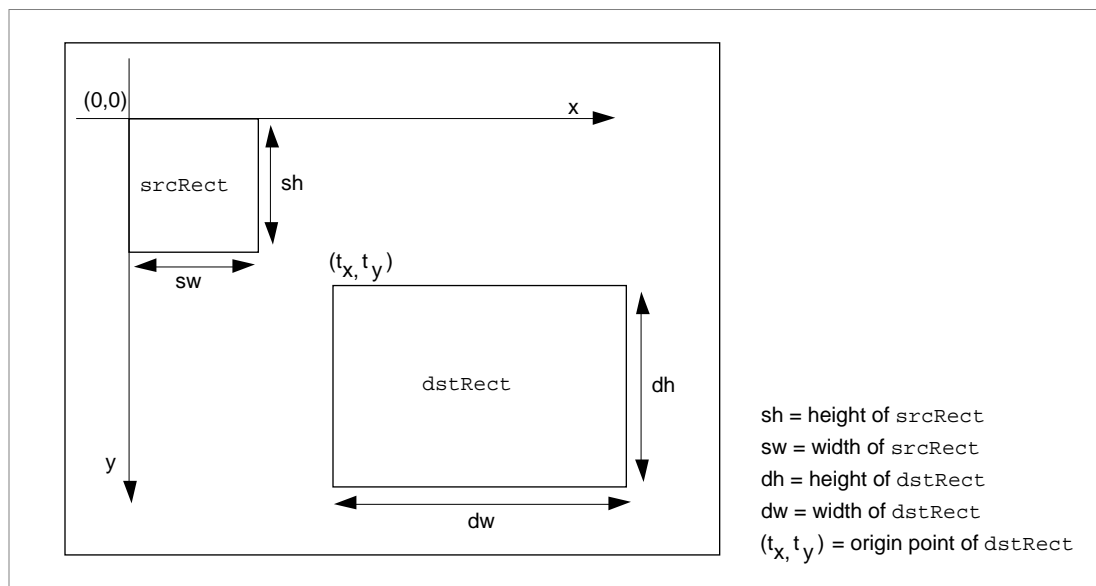
```
pascal void RectMatrix (MatrixRecord *matrix, Rect *srcRect,
                        Rect *dstRect);
```

<code>matrix</code>	Contains a pointer to a matrix structure. The <code>RectMatrix</code> function updates the contents of this matrix so that the matrix describes a transformation from points in the rectangle specified by the <code>srcRect</code> parameter to points in the rectangle specified by the <code>dstRect</code> parameter. The previous contents of the matrix are ignored.
<code>srcRect</code>	Contains a pointer to the source rectangle.
<code>dstRect</code>	Contains a pointer to the destination rectangle.

### DESCRIPTION

You specify the two rectangles; the function returns the appropriate matrix. Figure 2-43 shows how this matrix transforms the source image.

**Figure 2-43** Transforming an image with the `RectMatrix` function



Calling the `RectMatrix` function with the two rectangles shown in Figure 2-43 results in the matrix shown in Figure 2-44.

**Figure 2-44** Matrix created as a result of calling the `RectMatrix` function

$$\begin{bmatrix} \frac{dw}{sw} & 0 & 0 \\ 0 & \frac{dh}{sh} & 0 \\ t_x & t_y & 1 \end{bmatrix}$$

#### SEE ALSO

If you call the `TransformRect` function (described on page 2-348) and supply the matrix produced by the `RectMatrix` function along with the source rectangle you specified when you called the `RectMatrix` function, the result is equivalent to the destination rectangle you specified.

## MapMatrix

The `MapMatrix` function alters an existing matrix so that it defines a transformation from one rectangle to another, similar to the `MapRect` and `MapRegion` routines that are described in *Inside Macintosh: Imaging*.

```
pascal void MapMatrix (MatrixRecord *matrix, Rect *fromRect,
                      Rect *toRect);
```

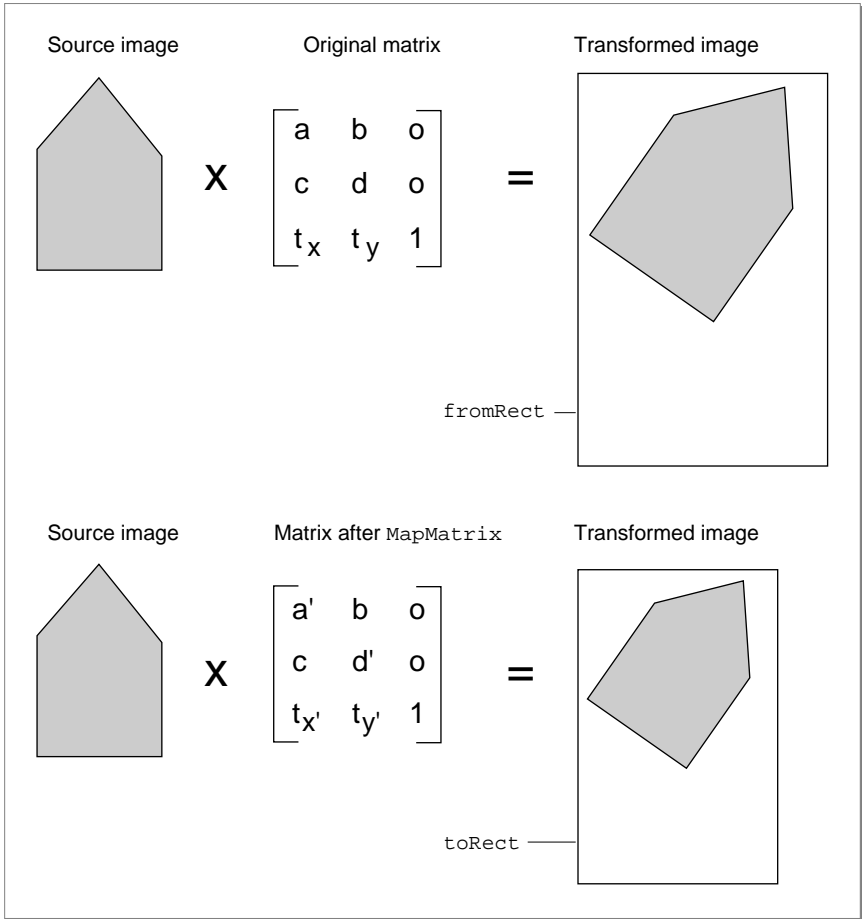
<code>matrix</code>	Contains a pointer to a matrix structure. The <code>MapMatrix</code> function modifies this matrix so that it performs a transformation in the rectangle specified by the <code>toRect</code> parameter that is analogous to the transformation it currently performs in the rectangle specified by the <code>fromRect</code> parameter.
<code>fromRect</code>	Contains a pointer to the source rectangle.
<code>toRect</code>	Contains a pointer to the destination rectangle.

DESCRIPTION

The `MapMatrix` function affects only the scaling and translation attributes of the matrix. This function is similar to `RectMatrix`, with the exception that `MapMatrix` concatenates the translation and scaling operations to the previous contents of the matrix, whereas `RectMatrix` first sets the matrix to the identity state.

Figure 2-45 shows how the matrix that you obtain from the `MapMatrix` function transforms a source image.

**Figure 2-45** Transforming an image with the `MapMatrix` function



SEE ALSO

You can create a matrix that maps one rectangle to another by calling the `RectMatrix` function, which is described in the previous section.



## Application-Defined Functions

---

This section describes the application-defined functions used with the Movie Toolbox. It is divided into the following topics:

- “Progress Functions” describes the functions that your application must assign to monitor the progress of the Movie Toolbox during long operations
- “Cover Functions” describes the functions that your application must use to perform custom processing whenever one of your movies covers a screen region or reveals a region that was previously covered
- “Error-Notification Functions” discusses the functions that your application must use to perform custom error-processing; you’ll find these functions particularly helpful when you’re debugging your program
- “Movie Callout Functions” describes the application-defined functions that the Movie Toolbox calls repeatedly while a movie preview is playing; you can use your movie callout function to stop the preview
- “File Filter Functions” provides details about the function that you can supply to filter the files that are displayed to the user in a dialog box
- “Custom Dialog Functions” supplies information about a function that lets you support the template in the custom dialog template that you specified with the `CustomGetFilePreview` function
- “Modal-Dialog Filter Functions” describes the functions that you can provide to support the custom dialog template you specified with your custom dialog function; your modal-dialog filter function gives you greater control over the interface presented to the user
- “Standard File Activation Functions” describes the functions that control the highlighting of any items whose shape is known only by your application
- “Callback Event Functions” discusses the callback events that you can ask the `CallMeWhen` function to schedule
- “Text Functions” describes a function through which you can specify operations on text and whether you want to display the text

## Progress Functions

---

Some Movie Toolbox functions can take a long time to execute. For example, creating a movie file that contains all of its data may be quite an involved process for a movie that has many large media structures. During these operations, your application should give the user some indication of the progress of the task. The Movie Toolbox allows you to monitor its progress on long operations with a progress function.

The Movie Toolbox calls your progress function at regular intervals during long operations. The Movie Toolbox determines whether to call your function based on the duration of the operation—your function will not be called unnecessarily. When it calls your function, the Movie Toolbox provides information about the operation that is

underway and its relative completion. You can use this information to display an informational dialog box to the user.

You assign a progress function to a movie by calling the `SetMovieProgressProc` function (described on page 2-155). You should assign your progress function when you open the movie; the Movie Toolbox will call your function when it is appropriate to do so. One progress function may support more than one movie. When the Movie Toolbox calls your function, it provides you with the movie identifier so that you can discriminate between various movies.

## MyProgressProc

---

Your progress function should support the following interface:

```
pascal OSErr MyProgressProc (Movie theMovie, short message,
                             short whatOperation,
                             Fixed percentDone, long refcon);
```

**theMovie** Specifies the movie for this operation. The Movie Toolbox sets this parameter to identify the appropriate movie.

**message** Indicates why the Movie Toolbox called your function. The following values are valid:

`movieProgressOpen`

Indicates the start of a long operation. This is always the first message sent to your function. Your function can use this message to trigger the display of your progress window.

`movieProgressUpdatePercent`

Passes completion information to your function. The Movie Toolbox repeatedly sends this message to your function. The `percentDone` parameter indicates the relative completion of the operation. You can use this value to update your progress window.

`movieProgressClose`

Indicates the end of a long operation. This is always the last message sent to your function. Your function can use this message as an indication to remove its progress window.

**whatOperation**

Indicates the long operation that is currently underway. The following values are valid:

`progressOpFlatten`

Your application has called the `FlattenMovie` or `FlattenMovieData` function (described on page 2-105 and page 2-107, respectively).

## Movie Toolbox

`progressOpInsertTrackSegment`

Your application has called the `InsertTrackSegment` function (described on page 2-262). The Movie Toolbox calls the progress function that is assigned to the movie that contains the destination track.

`progressOpInsertMovieSegment`

Your application has called the `InsertMovieSegment` function (described on page 2-257). The Movie Toolbox calls the progress function that is assigned to the destination movie.

`progressOpPaste`

Your application has called the `PasteMovieSelection` function (described on page 2-249). The Movie Toolbox calls the progress function that is assigned to the destination movie.

`progressOpAddMovieSelection`

Your application has called the `AddMovieSelection` function (described on page 2-250). The Movie Toolbox calls the progress function that is assigned to the destination movie. The Movie Toolbox calls the progress function that is assigned to the destination movie.

`progressOpCopy`

Your application has called the `CopyMovieSelection` function (described on page 2-248). The Movie Toolbox calls the progress function that is assigned to the destination movie.

`progressOpCut`

Your application has called the `CutMovieSelection` function (described on page 2-247). The Movie Toolbox calls the progress function that is assigned to the destination movie.

`progressOpLoadMovieIntoRam`

Your application has called the `LoadMovieIntoRam` function (described on page 2-140). The Movie Toolbox calls the progress function that is assigned to the destination movie.

`progressOpLoadTrackIntoRam`

Your application has called the `LoadTrackIntoRam` function (described on page 2-142). The Movie Toolbox calls the progress function that is assigned to the destination track.

`progressOpLoadMediaIntoRam`

Your application has called the `LoadMediaIntoRam` function (described on page 2-143). The Movie Toolbox calls the progress function that is assigned to the destination media.

## Movie Toolbox

<code>progressOpImportMovie</code>	Your application has called the <code>ConvertFileToMovieFile</code> function (described on page 2-93). The Movie Toolbox calls the progress function that is associated with the destination movie file. This flag is also used, as appropriate, for the <code>PasteHandleIntoMovie</code> functions (described on page 2-252).
<code>progressOpExportMovie</code>	Your application has called the <code>ConvertMovieFile</code> function (described on page 2-95). The Movie Toolbox calls the progress function that is associated with the destination movie. This flag is also used, as appropriate, for the <code>PutMovieIntoTypedHandle</code> function (described on page 2-253).
<code>percentDone</code>	Contains a fixed-point value indicating how far the operation has progressed. Its value is always between 0.0 and 1.0. This parameter is valid only when the message field is set to <code>movieProgressUpdatePercent</code> .
<code>refcon</code>	Reference constant value for use by your progress function. Your application specifies the value of this reference constant when you assign the progress function to the movie.

**DESCRIPTION**

Your progress function should return an error value. The Movie Toolbox examines this value after each `movieProgressUpdatePercent` message and before continuing the current operation. Set this value to a nonzero value, such as `userCanceledErr`, to cancel the operation; set it to `noErr` to continue.

**Cover Functions**

The Movie Toolbox 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 using cover functions.

There are two types of cover functions: those that are called when your movie covers a screen region, and those that are called when your movie uncovers a screen region, revealing a region that was previously covered. You can use a cover function to detect when a movie changes size.

Cover functions 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 an offscreen buffer. Cover functions that are called when your movie reveals a hidden screen region must redisplay the hidden region.

## Movie Toolbox

The Movie Toolbox sets the graphics world before it calls your cover function. Your function must not change the graphics world.

The Movie Toolbox provides default cover functions. When your movie uncovers a region, the default function that is called erases the movie's image by displaying the graphics port's background color and pattern. You can set the port's characteristics by calling the `SetMovieGWorld` function (described on page 2-159). When your movie covers a region, the default function that is called does nothing.

Use the `SetMovieCoverProcs` function (described on page 2-156) to set both types of cover functions.

## MyCoverProc

---

Your cover functions should support the following interface:

```
pascal OSErr MyCoverProc (Movie theMovie, RgnHandle changedRgn,
                          long refcon);
```

`theMovie`     Specifies the movie for this operation.

`changedRgn`

Contains a handle to the changed screen region.

`refcon`

Contains the reference constant that you specified when you defined the progress function.

### DESCRIPTION

Your cover function should always return an error value of `noErr`.

## Error-Notification Functions

---

The Movie Toolbox lets your application perform custom error notification. Your application must identify its custom error-notification function to the Movie Toolbox. Error-notification functions are particularly helpful when you are debugging your program.

The `SetMoviesErrorProc` function (described on page 2-86) allows you to identify your application's error-notification function in the `errProc` parameter.

## MyErrProc

---

The entry point to your error-notification function should take the following form:

```
pascal void MyErrProc (OSErr theErr, long refcon);
```

**theErr**        Contains the result code that the Movie Toolbox is going to place in the current error value.

**refcon**        Contains the reference constant value that you specified when your application called the `SetMoviesErrorProc` function.

## Movie Callout Functions

---

The `PlayMoviePreview` function (described on page 2-120) plays a movie's preview. You provide a pointer to a movie callout function in the `callOutProc` parameter.

The Movie Toolbox calls your movie callout function repeatedly while the movie preview is playing. You can use this function to stop the preview. If you do not want to assign a function, set the `callOutProc` parameter to `nil`.

## MyCalloutProc

---

Your movie callout function should present the following interface:

```
pascal Boolean MyCallOutProc (long refcon);
```

**refcon**        Contains the reference constant that you specified when you called the `PlayMoviePreview` function.

### DESCRIPTION

Your movie callout function returns a Boolean value. The Movie Toolbox examines this value before continuing. If your function sets this value to `false`, the Movie Toolbox stops the preview and returns to your application.

#### Note

If you call the `GetMovieActiveSegment` function (described on page 2-137) from within your movie callout function, the Movie Toolbox will have changed the active movie segment to be the preview segment of the movie. The Movie Toolbox restores the active segment when the preview is done playing. ♦

## File Filter Functions

---

A file filter function filters the files that are displayed to the user in a dialog box. You specify this function in the `fileFilter` parameter of the `SFGetFilePreview`, `StandardGetFilePreview`, and `CustomGetFilePreview` routines. If this parameter is not `nil`, `SFGetFilePreview` calls the function for each file to determine whether to display the file to the user. The `SFGetFilePreview` function supplies you with the information it receives from the File Manager's `GetFileInfo` routine (see *Inside Macintosh: Files* for more information).

## MyFileFilter

---

A file filter function whose address is passed to `SFGetFilePreview`, `StandardGetFilePreview`, or `CustomGetFilePreview` should have the following form.

```
pascal Boolean MyFileFilter (ParmBlkPtr parmBlock);
```

`parmBlock` A pointer to the parameter block associated with the files that are displayed to the user in this dialog box. For details, see *Inside Macintosh: Files*.

### DESCRIPTION

When `SFGetFilePreview`, `StandardGetFilePreview`, or `CustomGetFilePreview` is displaying the contents of a volume or folder, it checks the file type of each file and filters out files whose types do not match your application's specifications. If your application also supplies a file filter function, the Standard File Package calls that function each time it identifies a file of an acceptable type.

When your file filter function is called, it is passed, in the `parmBlock` parameter, a pointer to a catalog information parameter block. See the chapter "File Manager" in *Inside Macintosh: Files* for a description of the fields of this parameter block.

Your function evaluates the catalog information parameter block and returns a Boolean value that determines whether the file is filtered (that is, a value of `true` suppresses display of the filename, and a value of `false` allows the display). If you do not supply a file filter function, the Standard File Package displays all files of the specified types.

## Custom Dialog Functions

---

A dialog hook function handles user selections in a dialog box. A custom dialog function lets you support the template in the custom dialog template that you specified with the `CustomGetFilePreview` routine. This function corresponds to the File Manager's

CustomGetFile routine. See *Inside Macintosh: Files* for a complete description of the CustomGetFile routine.

You specify your dialog function in the `dlgHook` parameter of CustomGetFilePreview. You can use this parameter to support a custom dialog box function you have supplied by specifying a dialog template resource in your resource file. You specify the dialog template's resource ID with the `dlgID` parameter. If you are not supplying a custom dialog function, set this parameter to `nil`. For more information about using custom dialog functions with the CustomGetFile routine, see *Inside Macintosh: Files*.

## MyDlgHook

---

A dialog hook function should have the following form:

```
pascal short MyDlgHook (short item, DialogPtr theDialog,
                        Ptr myDataPtr);
```

<code>item</code>	The number of the item selected.
<code>theDialog</code>	A pointer to the dialog structure for the dialog box.
<code>myDataPtr</code>	A pointer to the optional data whose address is passed to CustomGetFilePreview.

### DESCRIPTION

You supply a dialog hook function to handle user selections of items that you added to a dialog box. If you provide a dialog hook function, CustomGetFilePreview calls your function immediately after calling the Dialog Manager's `ModalDialog` function. It passes your function the item number returned by `ModalDialog`, a pointer to the dialog structure, and a pointer to the data received from your application, if any.

Your dialog hook function returns as its function result an integer that is either the item number passed to it or some other item number. If your dialog hook function does not handle a selection, it should pass the item number back to the Standard File Package for processing by setting its return value equal to the item number. If your dialog hook function does handle the selection, it should pass back `sfHookNullEvent` or the number of some other pseudo-item.

### SEE ALSO

See *Inside Macintosh: Files* for another sample dialog hook function.



## Modal-Dialog Filter Functions

---

The `CustomGetFilePreview` routine presents an Open dialog box to the user and allows the user to view file previews. This function differs from `StandardGetFilePreview` in that you can provide a custom dialog template and functions to support your template. This function corresponds to the existing `CustomGetFile` routine.

You specify your modal-dialog filter function in the `filterProc` parameter. Your modal-dialog filter function gives you greater control over the interface presented to the user. See *Inside Macintosh: Files* for more information about using modal-dialog filter functions with `CustomGetFile`.

### Note

A modal-dialog filter function controls events closer to their source by filtering the events received from the Event Manager. The Standard File Package itself contains an internal modal-dialog filter function that maps keypresses and other user input onto the equivalent dialog box items. If you also want to process events at this level, you can supply your own filter function. ♦

## MyModalFilter

---

A modal-dialog filter function whose address is passed to `CustomGetFilePreview` should have the following form:

```
pascal Boolean MyModalFilter (DialogPtr theDialog,
                             EventRecord *theEvent,
                             short itemHit, Ptr myDataPtr);
```

<code>theDialog</code>	A pointer to the dialog structure of the dialog box.
<code>theEvent</code>	A pointer to the event structure for the event.
<code>itemHit</code>	The number of the item selected.
<code>myDataPtr</code>	A pointer to the optional data whose address is passed to <code>CustomGetFilePreview</code> .

### DESCRIPTION

Your modal-dialog filter function determines how the Dialog Manager's `ModalDialog` routine filters events. The `ModalDialog` routine retrieves events by calling the Event Manager's `GetNextEvent` routine. The Standard File Package contains an internal filter function that performs some preliminary processing on each event it receives. If you provide a modal-dialog filter function, `ModalDialog` calls your filter function after it calls the internal Standard File Package filter function and before it sends the event to your dialog hook function.

## Movie Toolbox

Your modal-dialog filter function returns a Boolean value that reports whether it handled the event. If your function returns a value of `false`, `ModalDialog` processes the event through its own filters. If your function returns a value of `true`, `ModalDialog` returns with no further action.

## SEE ALSO

See *Inside Macintosh: Files* for another sample modal-dialog filter function.

## Standard File Activation Functions

---

The `CustomGetFilePreview` function presents an Open dialog box to the user and allows the user to view file previews. This function differs from the `StandardGetFilePreview` function in that you can provide a custom dialog template and functions to support your template. The `CustomGetFilePreview` function corresponds to the File Manager's `CustomGetFile` routine.

You specify your activation function in the `activateProc` parameter. An activation function controls the highlighting of any items whose shape is known only by your application. See *Inside Macintosh: Files* for more information about standard file activation routines.

## MyActivateProc

---

An activation function should have the following form:

```
pascal void MyActivateProc (DialogPtr theDialog, short itemNo,
                           Boolean activating, Ptr myDataPtr);
```

`theDialog`    A pointer to the dialog structure of the dialog box.

`itemNo`       The number of the item selected.

`activating`    A Boolean value that specifies whether the field is being activated (`true`) or deactivated (`false`).

`myDataPtr`    A pointer to the optional data whose address is passed to `CustomGetFilePreview`.

## DESCRIPTION

Ordinarily, you need to supply an activation function only if your application builds a list from which the user can select entries. The Standard File Package supplies the activation routine for the file display list and for all `TextEdit` fields. You can also use the activation function to keep track of which field is receiving keyboard input, if your application needs that information.

## Movie Toolbox

Your application is responsible for removing the highlighting when one of its fields becomes inactive and for adding the highlighting when one of its fields becomes active. The Standard File Package can handle the highlighting of all `TextEdit` fields, even those defined by your application.

## Callback Event Functions

---

The `CallMeWhen` function (described on page 2-337) schedules a callback event. You specify the callback event in the `callbackProc` parameter.

## MyCallback

---

Your callback function must support the following interface:

```
pascal void MyCallbackProc (QTCallback cb, long refcon);
```

`cb` Specifies the callback event for the operation.

`refcon` Contains a reference constant value for your callback function.

## Text Functions

---

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

## MyTextProc

---

Your text function should have the following form:

```
pascal OSErr MyTextProc (Handle theText, Movie theMovie,
                        short *displayFlag, long refcon);
```

`theText` Contains a handle to the formatted text.

`theMovie` Specifies the movie for this operation.

## Movie Toolbox

`displayFlag`

Contains a pointer to one of the following flags, which specify how you want the text media handler to proceed when your function returns. The three possible return values for the flag are:

`txtProcDefaultDisplay`

Indicates that the media should follow the instructions of its own `displayFlag` constants.

`txtProcDontDisplay`

Tells the media not to display the text.

`txtProcDoDisplay`

Instructs the media to display the text regardless of the media's own `displayFlag` constants.

`refcon`

Contains the reference constant to your text function.