New Technical Notes

Macintosh

Developer Support

QT 01 - Inside Macintosh: QuickTime Addendum

QuickTime

Written by: Developer Technical Support and QuickTime EngineeringDecember 1994

This Tech Note is an addendum to the Inside Macintosh: QuickTime publication. It will contain technical details of QuickTime missing in the documentation, updated information, known problems, workarounds, bug fixes and similar information. The subtitles are based on the QuickTime Publication with the addition of new ones related to additional information not present in the documentation.

We assume that developers use QuickTime 1.6.1 or QuickTime 2.0, any older versions are no longer supported by DTS.

Table of Contents

MCSetClip and Clipping with the Movie Controller
Determining QuickDraw Video Media Pixel Depth
SetMovieDrawingCompleteProc
SetTrackGWorld
GetMovieCoverProcs
Access to Decompressed Images during Playback, GWorlds
Specifying Where a Movie is Pasted Using an Offset, GetTrackMatrix, SetTrackMatrix
Functions for Editing Movies
showUserSettingsDialog Flag
NewMovieFromScrap, Adding the Media Later

CHAPTER 2 - MOVIE TOOLBOX

Functions for Getting and Playing Movies

EnterMovies Use and Implementation

Q: The QuickTime 1.0 documentation says I can EnterMovies multiple times, as long as I balance each EnterMovies with ExitMovies. But the article in issue 13 of develop says, “Don’t call ExitMovies. ExitToShell does this for me.” So now the only call I have to use is EnterMovies. But this seems to destroy the balance of these routines. What should I do?

—

A: The way EnterMovies works is that it creates a new QuickTime environment if the current a5 world doesn’t have one already. If the current a5 world already has one from a previous EnterMovies call, then nothing is done except that a counter is incremented to keep track of the number of times EnterMovies has been called. The original reason for this was to account for DAs which use an application’s a5 world. By doing this, if an application has called EnterMovies already, the movie toolbox knows that it doesn’t have to reinitialize when the DA calls EnterMovies. Then, when the DA calls ExitMovies, it decrements the counter and as long as all EnterMovies and ExitMovies calls are balanced, the Movie Toolbox won’t dispose of the QuickTime world until the last ExitMovies call is made.

The reason we suggest not to call ExitMovies is that ExitToShell will automatically call ExitMovies for you. By doing so, you can avoid some of the problems that developers have had with disposing of movie structures improperly and in the wrong order. Letting ExitToShell do the final cleanup avoid these problems because the entire a5 world and heap is disposed of as well.

But, if you’ve nested EnterMovies and ExitMovies calls, this is what we recommend:

1. If you’re writing an application, call EnterMovies and don’t call ExitMovies. This way, any external that uses the same a5 world and calls EnterMovies and ExitMovies will simply increment a counter and then decrement the counter and do nothing else.

2. If you’re writing an external that runs under someone else’s application, you have two choices:

i. Call EnterMovies and never call ExitMovies. This causes QuickTime to initialize once during the first call to any of your externals.

ii. Call EnterMovies in the beginning of each external and ExitMovies at the end of each external. This can cause a lot of wasted CPU time if the main application (or someone else in the a5 world) didn't call EnterMovies first because each of the EnterMovies calls in your externals will require QuickTime to reinitialize. The reason for this is that you balance every EnterMovies and the corresponding ExitMovies will dispose of the QuickTime world. This can be very bad if you external gets called often. On the other hand, if your external has a initialization routine and close routine that is called before and after all your other routines, you can call EnterMovies in the initialization routine, and then call ExitMovies in your close routine. Of course, each of the routines called in between could call either EnterMovies and ExitMovies, or not call either of them at all. But, this depends on your implementation.

QuickTime Movie Toolbox *Globals* are Stored in System Heap

Q: According to the QuickTime Movie Toolbox documentation, "The Movie Toolbox maintains a set of global variables for every application using it." How much global memory is required? Our application is shy on global data space.

—

A: The information maintained is not kept with the application's global variables. The handle created by the EnterMovies call is stored in the system heap, not in the application heap. You don't have to worry about how much space to allocate in your application. This initialization does not affect your A5 world either.

`EnterMovies` initializes everything, including setting up the necessary data space and creating a handle to it.

Different A5 Worlds with QuickTime?

Q: Can we use a different A5 world with QuickTime? Our plug-in architecture uses A5 for global access, but we allow the A5 world to move. QuickTime doesn't seem to appreciate this and doesn't think that EnterMovies has been called after the A5 world moves. We currently work around this by locking down our A5 world but would rather not. Is locking down the A5 world even good enough?

—

A: You can use a different A5 world with QuickTime. QuickTime allocates a new set of state variables for each A5 world that's active when EnterMovies is called. However, since QuickTime uses A5 to identify each QuickTime client, if you move your plug-in's A5 world, QuickTime will no longer recognize that you've called EnterMovies for that client. So you can use a different A5 world, but you will have to lock the A5 world down.

How to Get Movie Frame Time Parameter for GetMoviePict

Q: How do I find the correct time values to pass to GetMoviePict, to get all the sequential frames of a QuickTime movie?

—

A: The best way to find the correct time to pass to get movie frames is to call the GetMovieNextInterestingTime routine repeatedly. Note that the first time you call GetMovieNextInterestingTime its flags parameter should have a value of nextTimeMediaSample+nextTimeEdgeOK to get the first frame.

For subsequent calls the value of flags should be nextTimeMediaSample. Also, the whichMediaTypes parameter should include only tracks with visual information, VisualMediaCharacteristic or 'eyes'. Check the Movie Toolbox chapter of the QuickTime documentation for details about the GetMovieNextInterestingTime call.

PutMovieIntoHandle and Data Forks

Q: I save PICTs to my document's data fork by writing the contents of the PicHandle. To save movies, do I convert the movie to a handle, and then save that as I would with PICTs? I just want the file references, not the data itself.

—

A: To save movies that are suitable for storage in a file, use PutMovieIntoDataFork. This function will store a movie in the data fork of a given file, and you could call NewMovieFromDataFork to reconstruct the movie for playback or editing.

You should also read the documentation regarding the Movie Toolbox FlattenMovie procedure, which creates a file that contains the 'moov' resource and the data all in the data fork. The advantage here is that the movie file you create using FlattenMovie can be read by any other QuickTime-capable application.

Embedding Movies (Multiple) Into a Macintosh File

Q: Is there a way to embed a QuickTime movie into a Macintosh file containing non-QuickTime stuff and get the Movie Toolbox to play the movie back correctly? If so, can we pass the same movie handle to QuickTime for Windows and get it to play back the same data from the same file?

—

A: To add QuickTime movie data to non-QuickTime files, just store the movie data in the file using FlattenMovieData with the flattenAddMovieToDataFork flag. Since FlattenMovieData will simply append to a data fork of a file, you can pass it any data file and it will append the movie data to that file. QuickTime doesn't care what's stored before or after the movie data, as long as you don't reposition the movie data within the data file. If you do, the movie references will be incorrect since they aren't updated when you edit the file. The returned movie (from FlattenMovieData) will properly resolve to that data file. You can then save this movie in the data fork with PutMovieIntoDataFork or in the resource fork with AddMovieResource. If the movie is saved in the data fork, it can be retrieved by both QuickTime and QuickTime for Windows with NewMovieFromDataFork.

You can, in fact, store multiple movies simply by calling FlattenMovieData and PutMovieIntoDataFork several times on the same file. Each FlattenMovieData call appends new data, assuming the createMovieFileDataCurFile flag isn't set.

See also the article Cross-Platform Compatibility and Multiple-Movie Files by John Wang in develop #17.

Clipping QuickTime Movie Posters

Q: Our application uses the movie poster as a still frame in a cell, similar to using a PICT. If a user sizes the cell width so that it's narrower than the poster, even though we clip the drawing to the cell size, QuickTime posters draw their full width, writing over whatever is in the way. Pictures clip through DrawPicture; why doesn't ShowMoviePoster stay within the clipping region?

—

A: ShowMoviePoster, as well as the movie and preview showing calls, uses the movie clipping characteristics rather than the destination port's clipping region. You must set the movie's clipping region to obtain the results you want. An easier way to do this is to get the picture for the poster by calling GetMoviePosterPict, and then simply use DrawPicture to display the poster. Because this is just a picture, the clipping region of the port is honored. This way you don't need different code for movies and pictures.

QuickTime File Audio Retrieval

Q: How can I retrieve audio from QuickTime files in 1-second chunks? I need a sound equivalent of GetMoviePict.

—

A: PutMovieIntoTypedHandle will take a movie or a single track from within the movie and convert it to a handle in memory of a specified type. This way you could take the sound track and convert it into a handle.

hintsHighQuality Flag

hintsHighQuality is a flag you may pass to the SetMoviePlayHints and SetMediaPlayHints routines. It specifies that the given movie or media should render at the highest quality. Rendering at highest quality may take considerably more time and memory. Therefore, this mode is typically not appropriate for real-time playback, but is very useful for re-compressing as it can generate higher quality images.

```
hintsHighQuality = 1<<8
```

The high-quality mode can be used with other media handlers as well. For example, the Video Media Handler turns off fast dithering and allows high-quality dithering.

QuickTime CFM PowerPlug Libraries, Availability, Weak Links

The Code Fragment Manager supports the concept of "soft" or "weak" linking. If a library is soft linked, then the Process Manager will go ahead and run your application even if the library is missing. This means that the application will not die even if a particular library is not installed, and the application could disable functionality based on what libraries are available or not.

MPW PowerPC tools include the MakePEF tool with an additional flag that specifies that the exported symbols are weak. In this case the runtime architecture will try to resolve the CFM library, but won't fail if it can't. You can define that the library has weak linking by adding a magic tilde (~) character at the end of the -l option to MakePEF. For example, to soft link to QuickTimeLib you would do the following:

```
MakePEF -l QuickTimeLib.xcoff=QuickTimeLib~ ...
```

You could also mark CFM libraries as weak using the Metrowerks PowerPC environment.

In the client program you can test that the CFM library was not loaded and for instance disable all functionality that depends on the CFM library (for example, no QuickTime CFM libraries present so "play movies" gets disabled).

Since the library only registers itself with Gestalt once, there's no way to unregister it if the user moves the library. This particular problem does not have any direct solutions, but there's an alternative way to determine if the library is loaded.

The solution is to check the address of one of the functions in the library before calling the library. The PowerPC Inside Macintosh documentation illustrates the technique:

```
extern int printf (char *, ...);
// ...
if (printf == kUnresolvedSymbolAddress)
       DebugStr("\printf is not available.");
else
       printf("Hello, world!\n");
```

QuickTime has a new Gestalt selector to determine whether it's safe to call the weak-linked library (gestaltQuickTimeFeatures). This function shows how to initialize QuickTime, for 68k and PowerPC:

```
Boolean    InitQuickTime(void)
{
       long         qtVersion;
       OSErr        anErr;
#ifdef powerc
       long         qtFeatures;
#endif

       anErr = Gestalt(gestaltQuickTime, &qtVersion);
       if (anErr != noErr)
              return false;       // no QT present

#ifdef powerc

// Test if the library is registered.
       anErr = Gestalt(gestaltQuickTimeFeatures, &qtFeatures);
   if ( !( (anErr == noErr)  &&  (qtFeatures & (1 << gestaltPPCQuickTimeLibPresent))
        )) // not true
         return false;
#endif

       anErr = EnterMovies();
       if ( anErr == noErr)
              return true;
       else
              return false;       // problems initializing QuickTime
}
```

Both QuickTime Gestalt selectors are being tested for in the PowerPC case. You need to test both that QuickTime is present and that the QuickTime library is present.

A simple test to verify that things work properly is to exclude the PowerPlug CFM library from the extension file, or move it out from the folder while the application that needs the library is running.

If the application is unable to load the CFM library due to lack of space, the application might mysteriously die later, so it's important to always check that the libraries are loaded and available.

Preroll Movies

It is of utmost importance that you preroll your movies using the PrerollMovie call. Failing to do so will introduce playback problems, especially when the movie starts. PrerollMovie will fill caches and buffers optimally to prevent initial playback stuttering.

Note that StartMovie will preroll the movie; also, the standard controller prerolls the movie whenever the user starts a movie using the keyboard or the mouse. In these situations, a possible second PrerollMovie call is redundant and will waste time and resources.

In all other cases, you should preroll the movie. For instance, it is your responsibility to call PrerollMovie if you are using SetMovieRate, or if you use McDoAction with mcActionPlay and a rate. Here's an example of how to use PrerollMovie:

```
OSErr DoPrerollMovie(Movie theMovie)
{
       TimeValue           aTimeValue;
       TimeValue           aMovieDur;
       Fixed               aPreferredRate;
       OSErr               anErr = noErr;

       aTimeValue    = GetMovieTime(theMovie, nil);
       aMovieDur     = GetMovieDuration(theMovie);
       aPreferredRate = GetMoviePreferredRate(theMovie);

       anErr = PrerollMovie(theMovie, aTimeValue, aPreferredRate);

       return anErr;
}
```

CustomGetFilePreview Problem - Missing 'dctb'

The system's pop-up CDEF has a problem that is short-circuited if a 'dctb' is present. The popup menu in the dialog that will navigate through folders is mis-positioned far to the right if you use CustomGetFilePreview when the "Show Preview" check box is unchecked. This happens when the DLOG/DITL resource is not associated with a 'dctb' resource, in other words, when the default color table is specified with ResEdit.

The workaround is to create a 'dctb' for the dialog.

Functions that Modify Movie Properties

QuickTime Track and Movie Sound Volume

Q: What do the values of a movie's or track's volume represent? Is there no way to make a track louder?

—

A: The volume is described as a small fract 8:8 and its values go from -1 to 1 with negative values as place holders. The maximum volume you can get is 0x0100 with the minimum being 0 (or any negative value). The advantage of using negative volumes is that you can turn off sound while maintaining the level of volume. For example, -1 and 0 both equate to no volume, but the -1 implies that 1 should be the volume when sound is turned back on, whereas the 0 does not.

The volume for a track is scaled to the movie's volume, and the movie's volume is scaled to the value the user specifies for the speaker volume using the Sound control panel. This means that the movie volume represents the maximum loudness of any track in the movie.

Note that starting with Sound Manager 3.0 you are able to increase the loudness above the maximum level using the shift key.

How to Get the First Video Frame

Q: Stepping through QuickTime movie video frames in the order they appear in the movie is simple using GetMovieNextInterestingTime, except for getting the first frame. If I set the time to 0 and rate to 1, I get the second frame, not the first. In addition, the video may start later than at 0. How do you suggest finding this first frame of video?

—

A: To get the first frame under the conditions you describe, you have to pass the flag nextTimeEdgeOK to GetMovieNextInterestingTime. What this flag does is make the call return the current interesting time instead of the next, if the current time is an interesting time. You need to do this because there's no way to go negative and then ask for the next interesting time.

MCSetClip and Clipping with the Movie Controller

Q: I use SetMovieDisplayClipRgn to set my movie clip, but the movie doesn't obey my clipping. Does the movie controller component ignore this clipping?

—

The controller uses the display clip for its own purposes, such as for badges. If you want to do clipping with the movie controller you must use MCSetClip. MCSetClip takes two regions. The first clips both the movie and the controller. The second clips just the movie, and is equivalent to the movie display clip. If both clips are set, the controller does the right thing and merges them as appropriate. If you don't want one or the other of the clips, set them to zero.

In general, if you are going to do something to a movie that is attached to a controller you must either do it through the controller, using the action calls, or you must call MCMovieChanged. Otherwise, the controller would need to constantly poll the movie to see if its state changed. Clearly this would be slow.

Determining QuickDraw Video Media Pixel Depth

Q: How do I get the pixel depth of the QuickTime video media for a given track?

—

A: To find the video media pixel depth, you'll need to retrieve the media's image description handle. You can use GetMediaSampleDescription to get it, but this routine needs both the video media and the track's index number. It's not obvious, but a media's type is identified by its media handler's type. Thus, you can walk through a movie's tracks by using its indexes until you find video media, at which point you have both the track index and video media.

The following sample code does the trick:

```
Media GetFirstVideoMedia(Movie coolMovie, long *trackIndex)
{
 Track    coolTrack = nil;
 Media    coolMedia = nil;
 long    numTracks;
 OSType  mediaType;
 numTracks = GetMovieTrackCount(coolMovie);
 for (*trackIndex=1; *trackIndex<=numTracks; (*trackIndex)++) {
   coolTrack = GetMovieIndTrack(coolMovie, *trackIndex);
   if (coolTrack) coolMedia = GetTrackMedia(coolTrack);
   if (coolMedia) GetMediaHandlerDescription(coolMedia,
     &mediaType, nil, nil);
   if (mediaType = VideoMediaType) return coolMedia;
 }
 *trackIndex = 0;  // trackIndex can't be 0
 return nil;      // went through all tracks and no video
}

short GetFirstVideoTrackPixelDepth(Movie coolMovie)
{
 SampleDescriptionHandle imageDescH =
   (SampleDescriptionHandle)NewHandle(sizeof(Handle));
 long    trackIndex = 0;
 Media    coolMedia = nil;
 coolMedia = GetFirstVideoMedia(coolMovie, &trackIndex);
 if (!trackIndex || !coolMedia) return -1;  // we need both
 GetMediaSampleDescription(coolMedia, trackIndex, imageDescH);
 return (*(ImageDescriptionHandle)imageDescH)->depth;
}
```

Note that QuickTime 2.0 has a new function called GetMovieIndTrackType that does most of the work described in the sample. GetMovieIndTrackType lets you search for all of a movie's tracks that share a given media type or media characteristic. See the QuickTime 2.0 SDK documentation for more details.

SetMovieDrawingCompleteProc

SetMovieDrawingCompleteProc lets you set a callback procedure that is called after a movie has drawn in one or more of its tracks. In this way, your application can be aware of when QuickTime has drawn frames and when it hasn’t. This information is very useful when combined with SetTrackGWorld (see below).

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

  theMovie          The Movie to set the proc on.
  proc              Your call back procedure, or nil to remove it.
  refCon            Value to pass to your callback procedure.

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

Errors:

```
invalidMovie        -2010  Your movie reference is bad.
```

SetTrackGWorld

SetTrackGWorld lets you force a track to draw into a particular GWorld. This GWorld may be different from that of the entire movie. After the track has drawn, it calls your transfer procedure to copy the track to the actual movie GWorld. When your transfer procedure is set, the current GWorld is set to the correct destination. You can also install a transfer procedure and set the GWorld to nil. This results in your transfer procedure being called only as a notification that the track has drawn—no transfer needs to take place.

```
pascal void SetTrackGWorld(Track theTrack, CGrafPtr port, GDHandle gdh,
TrackTransferProc proc, long refCon)

  theTrack          The track to set the proc to.
  port              The port for the track to draw to, or nil to use the movie's GWorld.
  gdh               GDevice associated with the port, or nil.
  proc              Returns pointer to your transfer procedure, or nil to remove it.
  refCon            Value to pass to your transfer procedure.

typedef pascal OSErr (*TrackTransferProc)(Track t, long refCon);
```

Errors:

```
invalidTrack        -2009  Your track reference is bad.

typedef struct {
       GWorldPtr    gw;
       GWorldPtr    efxTrack;
       GWorldPtr    tween;
       short        trackStat;
       Rect         dst;
       WindowPtr    wp;
} mSpfx;

typedef struct {
       Movie                    mv;
       MovieController          mctl;
       Rect                     mrect;
       mSpfx                    *mefx;
       GWorldPtr                backPict;
} mvInfo, *mvPtr;

/* these are the track transfer procedures, all they do is set a flag to */
/* indicate to the drawing completion proc that both tracks are ready */

pascal OSErr FrontTrackTransferProc(Track t, mSpfx *mfx)
{
       mfx->trackStat |= 1;             // first bit for the front, or main track
       return noErr;
}

pascal OSErr EfxTrackTransferProc(Track t, mSpfx *mfx)
{
       mfx->trackStat |= 2;             // second bit for the special effects track
       return noErr;
}

pascal OSErr MovieDrawingProc(Movie m, mvPtr mvp) {}

void SetUpMovieEffect(Movie m, WindowPtr wp)
{
       Track  t;
       mSpfx  *mfx;
       OSErr  err;
       Rect   bounds;
       mvPtr  mvi;
       long   numTracks;
```

```
	/* set up the transfer procedures for each track */
	/* track 1 is the main movie track */
	/* track 2 is the special effects track */
	t = GetMovieIndTrack(m,1);

	SetTrackGWorld(t, mfx->gw, nil, (TrackTransferProc)FrontTrackTransferProc,
			(long) mfx);
	t = GetMovieIndTrack(m,2);
	SetTrackGWorld(t, mfx->efxTrack, nil,
			(TrackTransferProc)EfxTrackTransferProc, (long) mf   /*
set up the routine that actually does the drawing */
	/* this routine is called after the movie toolbox draws all the tracks */ /*
into the offscreen GWorlds set up above */
	SetMovieDrawingCompleteProc(m, (MovieDrawingCompleteProcPtr)MovieDrawingProc,
				(long) mvi);
	GoToBeginningOfMovie(m);
}
```

GetMovieCoverProcs

GetMovieCoverProcs lets you retrieve the cover procedures that you set with SetMovieCoverProcs.

```
pascal OSErr GetMovieCoverProcs(Movie theMovie, MovieRgnCoverProc *uncoverProc,
	MovieRgnCoverProc *coverProc, long *refcon)

  Movie             Movie reference.
  MovieRgnCoverProc Returns the uncover proc for the movie.
  MovieRgnCoverProc Returns the cover proc for the movie.
  long              Returns the refcon for the cover procedures.
```

Errors:

```
invalidMovie        -2010  Your movie reference is bad.
```

Access to Decompressed Images during Playback, GWorlds

Q: Is there a mechanism that allows us to access each decompressed image prior to display during playback, so that we could manipulate the image data and then hand it back for display?

—

A: QuickTime 1.6.1 provides a function called SetTrackGWorld. SetTrackGWorld lets you force a track to draw into a particular GWorld. This GWorld may be different from that of the entire movie. After the track is drawn, it will call your transfer procedure to copy the track to the actual movie GWorld. When your transfer procedure is set, the current GWorld is set to the correct destination.

You could also install a transfer procedure and set the GWorld to nil. This results in your transfer procedure being called only as a notification that the track has drawn and that no transfer is taking place.

Inside your transfer procedure you could manipulate the image. Note that calling resource intensive or time consuming routines in your transfer procedure may have an adverse effect on the playback performance of the movie that is playing.

Here's an example of a transfer procedure that will keep a counter of number of times it has been called, and displays this number in the top left corner of the movie:

```
pascal OSErr myTrackTransferProc(Track t, long refCon)
{

       TransferDataHandle         myTDH = (TransferDataHandle)refCon ;
       GrafPtr                    theNewWorld ;
       GrafPtr                    movieGWorld ;
       PixMapHandle               offPixMap ;
       Rect                       movieBox ;
       static long                index = 1 ;
       CGrafPtr                   savedWorld ;
       GDHandle                   savedDevice ;
       Str255                     theString ;



       movieGWorld = (GrafPtr)((**myTDH).movieGWorld) ;
       theNewWorld = (GrafPtr)((**myTDH).trackGWorld) ;
       movieBox    = (**myTDH).movieRect ;

       offPixMap = GetGWorldPixMap( (GWorldPtr)theNewWorld ) ;
       (void) LockPixels( offPixMap ) ;

       GetGWorld( &savedWorld, &savedDevice );
       SetGWorld( (CGrafPtr)theNewWorld, nil ) ;

       MoveTo ( 15, 15 );
       NumToString ( index++, theString );
       DrawString ( theString );

       // copy the image from the offscreen port
       // into the movies port

       SetGWorld( savedWorld, savedDevice ) ;

       CopyBits(    &theNewWorld->portBits,
                           &movieGWorld->portBits,
                           &theNewWorld->portRect,
                           &movieBox,
                           srcCopy,
                           nil ) ;

       (void) UnlockPixels( offPixMap ) ;
}


//--------------------------------------------------------------------------
// define a structure to hold all the information we need in the transfer
// proc.

typedef struct {
       GWorldPtr    movieGWorld ;
       GWorldPtr    trackGWorld ;
       Rect         movieRect ;
} TransferData, *TransferDataPtr, **TransferDataHandle ;

//This has the original movie gWorld, the one we created for the track and a rect
```

```
// describing the movie.  You can set a movie up to use this in the following way:

       TransferDataHandle  myTDH = (TransferDataHandle)NewHandle( sizeof(
                                        TransferData )) ;
       Track        aTrack = GetFirstTrackOfType( aMovie, VideoMediaType ) ;
       short        trackDepth = GetFirstVideoTrackPixelDepth( aMovie ) ;

       if( myTDH == nil || aTrack == nil || trackDepth < 0)
              return ;

       GetTrackDimensions( aTrack, &width, &height ) ;

       trackDimensions.right = Fix2Long( width );
       trackDimensions.bottom = Fix2Long( height );

       // create the movie gWorld
       theErr = NewGWorld( &theNewWorld, trackDepth, &trackDimensions, nil,
                                  theNewWorldDevice, 0L ) ;
       CheckError( theErr, "\pCall to NewGWorld failed" );

       GetMovieGWorld( aMovie, &movieGWorld, nil ) ;

       (**myTDH).movieGWorld = movieGWorld ;
       (**myTDH).trackGWorld = theNewWorld ;

       GetMovieBox( aMovie, &movieBox ) ;
       (**myTDH).movieRect = movieBox ;

       SetTrackGWorld( aTrack, (CGrafPtr)theNewWorld, nil, myTrackTransferProc,
                           (long)myTDH ) ;
```

Specifying Where a Movie is Pasted Using an Offset, GetTrackMatrix, SetTrackMatrix

Q: When a user pastes a movie into a movie-controller movie, the added movie is inserted in the top left corner of the movie. Is there a way for the user to choose where the movie is pasted, and if not, how can I give the movie controller or Movie Toolbox an offset to use rather than have the editing operations use the top left corner?

—

A: When you paste a movie into a movie-controller movie, the movie controller is simply calling PasteMovieSelection to insert the source movie. All the characteristics of the movie are inserted, and therefore the movie is inserted in the top left corner of the movie. There's no easy way to specify an offset directly to the movie controller. If you want to change the offset of the pasted movie, you'll have to modify the movie yourself after the paste using Movie Toolbox commands. Once you're done changing the movie, be sure to call MCMovieChanged so that the movie controller updates correctly.

The actual modification is simple: call GetTrackMatrix, add your offset to the matrix, and call SetTrackMatrix. The difficulty is in determining which tracks to modify, since the paste may either create a new track or use an existing one. We recommend doing this by gathering all track IDs before the paste, and then comparing with the track IDs after the paste. Since most movies these days have just a few tracks, this shouldn't require much overhead. (But be warned: some movies do have a lot of tracks!) To get the track information, you can call GetMovieTrackCount and GetMovieIndTrack.

One last idea: If you don't mind changing the source movie, an alternative is to simply offset the source movie before the paste.

Functions for Editing Movies

showUserSettingsDialog Flag

showUserSettingsDialog is a new flag. When using either PasteHandleIntoMovie or ConvertFileToMovieFile to import data into a movie, you can now set the showUserSettingsDialog flag. This displays the user settings dialog box for that import operation, if there is one. For example, when importing a picture, this would cause the Standard Compression dialog box to be displayed so the compression method could be selected.

```
showUserSettingsDialog = 2
```

NewMovieFromScrap, Adding the Media Later

Q: When my application creates a new media (of text type in this case) for a new track in a movie created with NewMovieFromScrap, the dataRef and dataRefType should be set to nil, according to the QuickTime documentation. The problem is that later I want to edit that media (adding a text sample to it, for example), but BeginMediaEdits returns the noDataHandler error(no data handler found). I assume I can get around that by first saving the movie to a file, but this seems slimy since the movie won't end up on disk in the end. Any suggestions for a better approach?

—

A: You're correct — BeginMediaEdits complains if the movie has been created with NewMovieFromScrap. Unfortunately, BeginMediaEdits doesn't think memory-based movies are on a media that will support editing. The workaround is to store the movie in a temporary file until you're finished editing it.

When you call NewTrackMedia, pass an alias to a new file in the dataRef parameter instead of nil. Passing nil (the usual approach) indicates that the movie's default data reference should be used, but because your movie came from the scrap and not a file, it has no data reference — hence the error you're getting. By the way, using the handle data handler in QuickTime 2.0 you can create a movie entirely in memory.

Sound Compression, IMA 4:1

Currently the only way to compress sounds using IMA 4:1 compression is to use the Sound Converter tool that is available in the QuickTime SDK.

Media Functions

Base Media Handler

New Extended Features in the Base Media Handler

Three new calls and a new flag extend the Base Media Handler interface. These features provide higher quality movie playback, but incur a performance penalty. The Text Media Handler takes advantage of these new calls and provides built-in support for anti-aliased text. It is achieved through a playback hint to the base media handler, which the Apple Text Media Handler derives. This hint, hintsHighQuality, has been discussed in the “Movie Toolbox Enhancements” section earlier in this Note.

The MediaSetHints and MediaGetOffscreenBufferSize routines were added to the Derived Media Handler interface to support high-quality mode. Since the Apple Text Media Handler derives the base media handler, it can use these new calls to support anti-aliased text.

MediaGetOffscreenBufferSize

MediaGetOffscreenBufferSize determines the dimensions of the offscreen buffer. Before the Base Media Handler allocates an offscreen buffer for your Derived Media Handler, it calls your MediaGetOffscreenBufferSize routine. The depth and color table used for the buffer are also passed. When this routine is called the bounds parameter specifies the size that the Base Media Handler intends to use for your offscreen by default. You can modify this as appropriate before returning. This capability is useful if your media handler can draw only at particular sizes. It is also useful for implementing anti-aliased drawing as you can request a buffer that is larger than your destination area and have the Base Media Handler scale the image down for you.

```
pascal ComponentResult MediaGetOffscreenBufferSize (ComponentInstance ci, Rect *bounds,
       short depth, CTabHandle ctab)

ci                  Component instance of a Base Media Handler.
bounds              The boundaries of your offscreen buffer.
depth               Depth of the offscreen.
ctab                Color table associated with offscreen. You can set it to nil.
```

Errors:

```
badComponentInstance       0x80008001   Get a new component instance.
```

MediaSetHints

MediaSetHints implements the appropriate behavior for the various media hints such as scrub mode and high-quality mode. When an application calls SetMoviePlayHints or SetMediaPlayHints, your media handler's MediaSetHints routine is called for each media in the movie.

```
pascal ComponentResult MediaSetHints (ComponentInstance ci, long hints)

  ci                Component instance of a Base Media Handler.
  hints             All hint bits that currently apply to the given media.
```

Errors:

```
badComponentInstance       0x80008001   Get a new component instance.
```

MediaGetName

MediaGetName lets you retrieve the name of the media type. For example, the Video Media Handler will return the string "Video."

```
pascal ComponentResult MediaGetName(MediaHandler mh, Str255 name, long requestedLanguage,
       long *actualLanguage )

  mh                The Base Media Handler instance.
  name              The name of the media type.
  requestLanguage   Language you want it to return name in.
  actualLanguage    Language it returns the name in.
```

Errors:

```
badComponentInstance       0x80008001   Get a new component instance.
```

Text Media Handler

New Display Flags

The display flags control the behavior of the Text Media Handler. The Text Media Handler is responsible for rendering the text. These flags provide additional control over the rendering process. To change the Text Media Handler's behavior with these flags, you will normally add these flags to each text sample. When the Text Media Handler reads each sample, it will also read the associated flags. The Text Media Handler will then adjust its behavior according to the display flag.

To add a text sample to the media, you use the routines AddTESample and AddTextSample. To add display flags to a text sample, you pass them in the displayFlags parameter of these routines.

```
enum {
dfContinuousScroll  = 1<<9,
dfFlowHoriz         = 1<<10,
dfDropShadow        = 1<<12,
dfAntiAlias         = 1<<13,
dfKeyedText         = 1<<14
};
```

dfContinuousScroll is a display flag that tells the Apple Text Media Handler to let new samples cause previous samples to scroll out. dfScrollIn and/or dfScrollOut must also be set for this to take effect.

dfFlowHoriz is a display flag that tells the Apple Text Media Handler to let horizontally scrolled text flow within the text box. This behavior contrasts with letting text flow as if the text box had no right edge.

dfDropShadow is a display flag that tells the Apple Text Media Handler to support true drop shadows. Using SetTextSampleData, the position and translucency of the drop shadow is under application control.

dfAntiAlias is a display flag that tells the Apple Text Media Handler to attempt to display text anti-aliased. While anti-aliased text looks nicer, it incurs a significant performance penalty.

dfKeyedText is a display flag that tells the Apple Text Media Handler to render text over the background without drawing the background color. This technique is otherwise known as "Masked Text."

findTextUseOffset is a new find text flag that instructs FindNextText to look at the value pointed to by the offset parameter and start the search at that offset into the text sample indicated by startTime. This allows you to continue a text search from within a given sample, so that multiple occurrences of the search string can be found within a single sample.

```
findTextUseOffset = 16
```

SetTextSampleData

SetTextSampleData allows you to set values prior to calling AddTextSample or AddTESample. Two types are currently supported: dropShadowOffsetType and dropShadowTranslucencyType. The first type, dropShadowOffsetType, is the drop shadow offset. Pass the address of a point for the data parameter. dropShadowTranslucencyType is the drop shadow translucency. Pass a value from 0 to 255, where 0 is the lightest and 255 is the darkest.

```
       #define      dropShadowOffsetType             'drpo'
       #define      dropShadowTranslucencyType 'drpt'

pascal ComponentResult SetTextSampleData(MediaHandler mh, void *data, OSType
       dataType)

mh           Reference to the Text Media Handler. Could use GetMediaHandler.
data         Pointer to data, defined by dataType parameter.
dataType     Sets the type of data in the handle. For now, either 'drpo' or 'drpt'.
```

Errors:

```
badComponentInstance       0x80008001    Your media reference is bad.
```

The following sample code snippet demonstrates the use of `SetTextSampleData`.

```
short  trans = 127;
Point  dropOffset;
MediaHandler mh;

dropOffset.h = dropOffset.v = 4;
SetTextSampleData(mh,(void *)&dropOffset,dropShadowOffsetType);
SetTextSampleData(mh,(void *)&trans,dropShadowTranslucencyType);
```

Be sure to turn on the dfDropShadow display flag when you call AddTextSample or AddTESample.

If you pass nil for textColor and/or backColor parameters in AddTextSample or AddTESample, they default to black (for textColor) and white (for backColor).

Matrix Functions

QuickTime Rotation and Skew aren't Implemented

Q: We can't apply the rotation and skew effects to a QuickTime 1.5 movie. We've created an identity matrix, applied RotateMatrix to the matrix, set the matrix to the movie using SetMovieMatrix, and played the movie. The movie didn't rotate but the movieRect rotated and the movie scaled to the movieRect. Is there anything wrong with what we're doing?

—

A: Rotation and skew will give you correct results for matrix operations but they haven't been implemented into QuickTime movie playback yet. Scaling and offset transformations now work with movies and images; rotation and skew are important future directions. Meanwhile, you can accomplish rotation and skewing by playing a movie to an off-screen GWorld and then use QuickDraw GX or your own graphics routines to display the rotated or skewed off-screen GWorld.

Status of Rotating Matrix Support

Q: What's the status of RotateMatrix and its use with SetMovieMatrix and SetTrackMatrix?

—

A: RotateMatrix works fine. But rotating matrixes are not supported for movies or images. So, although RotateMatrix will give you the correct mathematical result, unless you are using the matrix to transform something else (as with TransformFixedPoints) it has little use.

Rotation is a very important direction that is sure to get more attention in the future.

CHAPTER 3- IMAGE COMPRESSOR MANAGER

Image Compression Manager Functions

Creating Thumbnail PICTs

Q: How can I display the thumbnail of the `PICT` instead of some generic icon when I create QuickTime `PICT` files? This would really help with distinguishing files when someone wanted to create a movie and had a lot of these PICTs around.

—

A: You need to follow these four steps:

1. Get the thumbnail. You can use either the MakeThumbnailFromPicture or MakeThumbnailFromPictureFile routines as listed in the ImageCompression interface file. It will pass back the PicHandle for the thumbnail. To install into the Finder, you need icon resources (ICN#, ics#, icl8, ics8, icl4, ics4).

2. Make the thumbnail into a 'icsx' format to store it as a resource. (Please see MakeIcon on the Developer CD. It is not modified for pichandles so you may have to add a DrawPicture. Basically, you need to create a GWorld and create the appropriate 16- or 32-bit image.)

3. Add icons to resource. You can use the basic Resource Manager's WriteResource and AddResource calls to add the resource.

4. Set Finder bits: Stuff icon resources into the file itself with resource ID kCustomIconResource, and set the hasCustomIcon bit.

```
{ myCInfoPBRec.ioFlFndrInfo.fdFlags := BOR(myCInfoPBRec.ioFlFndrInfo.fdFlags, $0400) }.
```

DecompressSequenceBeginS

DecompressSequenceBeginS allows you to pass a compressed sample so the codec can do preflighting before the first DecompressSequenceFrame.

```
pascal OSErr DecompressSequenceBeginS(ImageSequence *seqID, ImageDescriptionHandle
       desc, Ptr data, CGrafPtr port, GDHandle gdh, const Rect *srcRect,
       MatrixRecordPtr matrix, short mode, RgnHandle mask, CodecFlags flags, CodecQ
       accuracy, DecompressorComponent codec)
```

seqID	Contains a pointer to a field to receive the unique identifier for this sequence returned by the CompressSequenceBegin function.
desc	Contains a handle to the image description structure that describes the compressed image.
port	Points to the graphics port for the destination image.
gdh	Contains a handle to the graphics device record for the destination image.
srcRect	Contains a pointer to a rectangle defining the portions of the image to decompress.
matrix	Points to a matrix structure that specifies how to transform the image during decompression.
mode	Specifies the transfer mode for the operation.
mask	Contains a handle to the clipping region in the destination coordinate system.
flags	Contains flags providing further control information.
accuracy	Specifies the accuracy desired in the decompressed image.
codec	Contains compressor identifier.

New Monitor Related Playback Calls

Three additional calls—GDHasScale, GDGetScale, GDSetScale—allow applications to zoom a monitor. They are considered low-level calls (comparable to SetEntries) that should be used only when playing back QuickTime movies in a controlled environment with no user interaction. Also, because this capability is not present on all machines, applications should not depend on its availability.

The new calls provide a standard way for developers to access the resizing abilities of a user's monitor for playback. Effectively, this allows you to have full screen Cinepak playback on low-end Macintosh computers.

Hardware 200 percent resize is currently available only on the Macintosh LC II, IIvx, IIvi, Performa 400, Performa 600, and Color Classic in 16-bit (thousands of colors) display mode on the 12-inch (512 x 384 pixels) monitors. In the future, other graphic devices may take advantage of it.

To implement this functionality, the Image Compression Manager actually makes calls to the video driver for the given device. Video card manufacturers interested in supporting this functionality in their cards should send an AppleLink to DEVSUPPORT (Internet: DEVSUPPORT@applelink.apple.com) for more information.

GDHasScale

GDHasScale returns the closest possible scaling that a particular screen device can be set to in a given pixel depth. It returns scaling information for a particular GDevice for a requested depth. It allows you to query a GDevice without actually changing it. For example, if you specify 0x20000, but the GDevice does not support it, GDHasScale will return with noErr, and a scale of 0x10000. Remember, it checks for a supported depth, so your requested depth must be supported by the GDevice. GDHasScale references the video driver through the graphics device structure.

For multiple screens, see "Multiple Screens Revealed" in *develop* #10 to find out how to walk the GDeviceList.

```
pascal OSErr GDHasScale(GDHandle gdh,short depth,Fixed *scale)

  gdh               A handle to a screen graphics device.
  depth             Pixel depth of screen device. Use this field to specify which pixel
                    depth scaling information should be returned for.
  scale             A pointer to a fixed point scale value. On input, this field should
                    be set to the desired scale value. On output, this field will
                    contain the closest scale available for the given depth. A scale of
                    0x10000 indicates normal size, 0x20000 indicates double size, and so
                    on.
```

Errors:

```
  cDepthErr         The requested depth is not supported.
  cDevErr           Not a screen device.
  controlErr        Video driver can not respond to this call.
```

GDGetScale

GDGetScale returns the current scale of the given screen graphics device.

```
pascal OSErr GDGetScale(GDHandle gdh,Fixed *scale,short *flags)

  gdh               A handle to a screen graphics device.
  scale             Pointer to a fixed point field to hold the scale result.
  flags             Pointer to a short integer. It returns the status parameter flags
                    for the video driver. For now, 0 is always returned in this field.
```

Errors:

```
cDevErr             Not a screen device.
controlErr          Video driver can not respond to this call.
```

GDSetScale

GDSetScale sets a screen graphics device to a new scale.

```
pascal OSErr GDSetScale(GDHandle gdh,Fixed scale,short flags)

  gdh               A handle to a screen graphics device.
  scale             A fixed point scale value.
  flags             Always pass 0.
```

Errors:

```
cDevErr             Not a screen device.
controlErr          Video driver can not respond to this call.
```

Using QuickTime Dither Tables in a Codec

Q: How can I use QuickTime fast dither tables provided by the Image Compression Manager to write a codec? I haven't been able to find any documentation on how to access and use them. Are these tables available?

—

A: For QuickTime 1.0 you could use the MakeDitherTable and DisposeDitherTable calls in ImageCompression.h. The calls were taken out for QuickTime 1.5 because the format is likely to change and your code would break in the future. The current dither table format isn't available for that reason, though the documentation on the QuickTime 1.0 CD describes the calls, if that helps.

You can use QuickTime to perform the dithering. If you do use QuickTime, you could draw the image in an off-screen GWorld, using the DrawPictureFile with the dither flag set, and then compress it with your codec.

How to Tell Whether a Picture is QuickTime-Compressed

Q: How can I tell whether or not a picture is QuickTime-compressed?

—

A: The key to your question is "sit in the bottlenecks." If the picture contains any QuickTime-compressed images, the images will need to pass through the StdPix bottleneck. This is a new graphics routine introduced with QuickTime. Unlike standard QuickDraw images, which only call StdBits, QuickTime-compressed images need to be decompressed first in the StdPix routine. Then QuickDraw uses StdBits to render the decompressed image. So, swap out the QuickDraw bottlenecks, and put some code in the StdPix routine. If it's called when you call DrawPicture, you know you have a compressed picture. To determine the type of compression, you can access the image description using GetCompressedPixMapInfo. The cType field of the ImageDescription record will give you the codec type. See the Snippets: Imaging: Graphics: CollectPictColors snippet and page 46-47 of develop Issue 13 for further reference on swapping out the bottlenecks.

QuickTime Fills in Image Descriptor when Data is Compressed

Q: When I send compressed images over Ethernet, CompressSequenceBegin doesn't fill in the ImageDescription, which is needed at the other end of the conference link to DecompressSequenceBegin. Is this a bug?

—

A: CompressSequenceBegin doesn't actually modify the handle that you pass. Instead, QuickTime makes a note of the handle that's passed and doesn't actually modify the contents until the first call that actually compresses data, such as CompressSequenceFrame. At that point, the handle will be changed.

If you can postpone dealing with the image descriptor until after the first call that compresses data, whatever you are writing should work just fine.

Decompressing to Partial window: Bug & Workaround

Q: Under System 7, decompressing directly to a window that is partially "off the screen" (that is, not completely visible) results in a -50 (invalid param) QuickTime error. We can special case when windows are off the screen and decompress into an offscreen GWorld but we would prefer a fix to either QuickTime or System 7.

—

A: The problem you are having is due to a bug in the Image Compression Manager. It fails to clear QDError when starting a decompression job and later checks it to see if it is OK to continue the operation. Something else is setting QDErr and your call fails.

The solution that you can implement now consists of clearing QDErr before calling any of the decompression routines. You can accomplish this by calling QDError (which clears the error after it passes the current value to you) or zeroing the low mem QDerr (0xD6E) by hand.

Future versions of QuickTime will have the fix and will not require that you work around the problem.

Problems With Matrixes and FDecompressImage, codecUnimpErr

Q: We are having problems related to decompressing a file compressed in the JPEG format using FDecompressImage. If a valid MatrixRecordPtr is passed to the routine, it returns an error -84962 (codecUnimpErr). If a parameter of NULL is passed as the MatrixRecordPtr, the routine works fine? Are matrix operations unsupported with the JPEG codec?

—

A: FDecompressImage only handles translation and scaling matrixes, so please check to see whether your matrix is either a translation of scaling matrix. Any other matrix types are reported back with the message codecUnimpErr.

When you specify NULL as the matrix you will get an identity matrix.

ImageDescriptionHandles, JPEG Files

Here are two ways to get access to an ImageDescriptionHandle for a JPEG file:

Case 1: You created the file in the first place, and this file is for the Macintosh platform only.
In this case you could store the ImageDescriptionHandle in a resource, and when you want to decompress the file, read this resource before doing the decompression operation.

Case 2: You read an arbitrary JPEG file generated by any possible platform. The QuickTime SDK CD has an example of a JFIF translator, JFIF is the interchange format of JPEG files across computer platforms. Check out the ScanJPEG function in this sample to see how to scan the file for the image description information that you could then later use in FDecompressImage.

PICT, QuickTime-Compressed Testing

If the picture contains any QuickTime-compressed images, the images will need to pass through the StdPix bottleneck. This is a new graphics routine introduced by QuickTime. Unlike standard QuickDraw images, which only call StdBits, QuickTime-compressed images need to be decompressed first in the StdPix routine. QuickDraw uses StdBits to render the decompressed image. Swap out the QuickDraw bottlenecks and place code in the StdPix routine. If this code is called when you call DrawPicture, you know you have a compressed picture. To determine the type of compression, you can access the image description using GetCompressedPixMapInfo. The cType field of the ImageDescription record will give you the codec type.

See the CollectPictColors snippet and “Inside QuickTime and Component-Based Managers” in develop Issue 13, specifically pages 46 and 47, for more information on swapping out the bottlenecks.

NIM Errata: OpCode Skipping With Size

The third bullet on page 3-27 (Inside Macintosh: QuickTime) specifies: "...Its size is included in the size for the main opcode, hence it is not included if the QuickTime opcode is skipped."

However, if the QuickTime opcode is skipped, the sub-opcode is still included (when DrawPicture is called). Its size does not include the sub-opcode.

Application-Defined Functions

SetSequenceProgressProc

SetSequenceProgressProc allows you to set a progress procedure on a Compression or Decompression Sequence, just as in the past you could have a progress procedure when compressing or decompressing a still image.

```
pascal OSErr SetSequenceProgressProc(ImageSequence seqID, ProgressProcRecord
      *progressProc)

seqID        Sequence identifier.
progressProc Pointer to a record containing information about the application's
             progress proc.
```

CHAPTER 4 -MOVIE RESOURCE FORMATS

'LOOP' and User Data Atoms

It is often desirable for an application to preserve the window position and looping state of a movie. This chapter describes the “Apple Sanctified” method of doing this using user data atoms.

User data atoms allow applications to store custom information which can be easily accessed using QuickTime Movie Toolbox calls. These user data atoms are text or data which can be associated and stored in any movie, track, or media. A reference to the list of user data atoms for each of these locations can be accessed with the following routines: GetMovieUserData(), GetTrackUserData(), and GetMediaUserData(). Once a reference to a list of user data atoms is obtained, an application can store, retrieve, and manage items in the list using the following routines: GetNextUserDataType(), CountUserDataType(), AddUserData(), GetUserData(), RemoveUserData(), AddUserDataText(), GetUserDataText, and RemoveUserDataText(). A complete description of these routines can be found in *Inside Macintosh: QuickTime* in the “Working With Movie User Data” section of chapter 2.

Getting 'LOOP'-y

MoviePlayer™ has defined two movie data atoms which are used to indicate looping and window location which applications can implement for compatibility with MoviePlayer™. They are:

'LOOP' If this 4 byte user data atom exists in the movie's user data list, then looping is performed according to its value: 0 for normal looping and 1 for palindrome looping

'WLOC' Handle to a point record indicating the last saved window position

Another variation on this which originated before MoviePlayer™ that applications should be aware of, is the following:

'LOOP' If this zero length data atom exists in the

movie's user data list, then normal looping is performed.

In summary, if a 'LOOP' atom exists, then looping should be performed. If the returned data is a long integer of value 1, then palindrome looping should be performed. Normal looping should be performed if data returned is of zero length or if the returned data is a long integer of value 0.

Sample Code

The following example demonstrates how to get looping information from a movie:

```
short        loopInfo;    //  0=no looping,1=normal looping,2=palindrome
                          //  looping

Handle       theLoop;
Movie        theMovie;
UserData     theUserData;

loopInfo = 0;
theLoop = NewHandle(0);
theUserData = GetMovieUserData(theMovie);
if (CountUserDataType(theUserData, 'LOOP')) {
    loopInfo = 1;
    GetUserData(theUserData, theLoop, 'LOOP', 1);
    if (GetHandleSize(theLoop))
        if ((** (long **) theLoop) == 1)
            loopInfo = 2;
}
```

The following example demonstrates how to add a looping atom to a movie to indicate that user has selected looping:

```
Handle       theLoop;
Movie        theMovie;
UserData     theUserData;
short        theCount;

theLoop = NewHandle(sizeof(long));
(** (long **) theLoop) = 0;
theUserData = GetMovieUserData(theMovie);
theCount = CountUserDataType(theUserData, 'LOOP');
while (theCount--)
    RemoveUserData(theUserData, 'LOOP', 1);
AddUserData(theUserData, theLoop, 'LOOP');
```

The following example demonstrates how to remove a looping atom from a movie to indicate that looping is not selected:

```
Movie        theMovie;
UserData     theUserData;
short        theCount;

theUserData = GetMovieUserData(theMovie);
theCount = CountUserDataType(theUserData, 'LOOP');
while (theCount--)
    RemoveUserData(theUserData, 'LOOP', 1);
```

Inside the User Data Atom

Those of you who parse user data atoms directly by accessing the 'moov' handle rather than with the appropriate movie toolbox calls, will notice a trailing long integer of value 0 after all user data atoms in the list. This is required for backward compatibility with QuickTime 1.0 which has a bug that requires the trailer. The size of the 'udta' atom does reflect this extra trailing long integer. QuickTime 1.0 and future versions will automatically handle this when manipulating user data atoms with the movie toolbox calls.

Saving a Movie's Active Selection

Q: I have a movie with two video tracks: Track #1—Enabled, Duration=10; Track #2—Disabled, Duration=20. I set the movie's active segment to (0, 10) and saved the movie resource to a movie file. When I open the movie in MoviePlayer, the movie is played for the time value of 20, ignoring the active segment and there being one enabled track with a duration of 10. Short of creating a new movie from a selection of the above mentioned movie, is there any way to get MoviePlayer to do what I'd expect and not ignore the active segment and play past the enabled track's duration?

—

A: The active selection isn't saved along with a movie. Therefore, no application will be able to restore and play back the active segment. You'd have to create a new movie from the selection in order to get MoviePlayer or any other application to play that selection only. If users will be using your application to play back the movie, you could store information regarding the active segment in a user data atom inside the movie. You could then have your application load the user data atom if it exists inside a movie when it's opened, to restore the selection. That would be the only way to save the active movie selection.

QuickTime Dependent File Format

Introduction

Dependent files are files that reference other files or are referenced themselves by other files. In this note, methods of working with QuickTime dependent files, as well as QuickTime's dependent file format are documented.

This note discusses issues mainly applicable to file management software. Most applications rely on the Finder for file management techniques such as copying, deleting, and so on. If your application does not delete or copy files with signatures other than your own and does not work with QuickTime files, then this note probably does not concern you. On the other hand, your application may already create dependent files, and you may wish to adopt QuickTime's method of tracking them.

Dependent File Overview

A QuickTime movie file may reference more than one file. In a common scenario, the movie's data may be stored in one file while the movie's resource may be stored in another. In fact, a QuickTime movie file may reference data stored in several files. For example, sound might be stored in one file, the video in another, and the movie resource itself in yet another. Logically, though, these files belong together and, thus, dependent files were created.

Dependent files use customized aliases to refer to the other files.

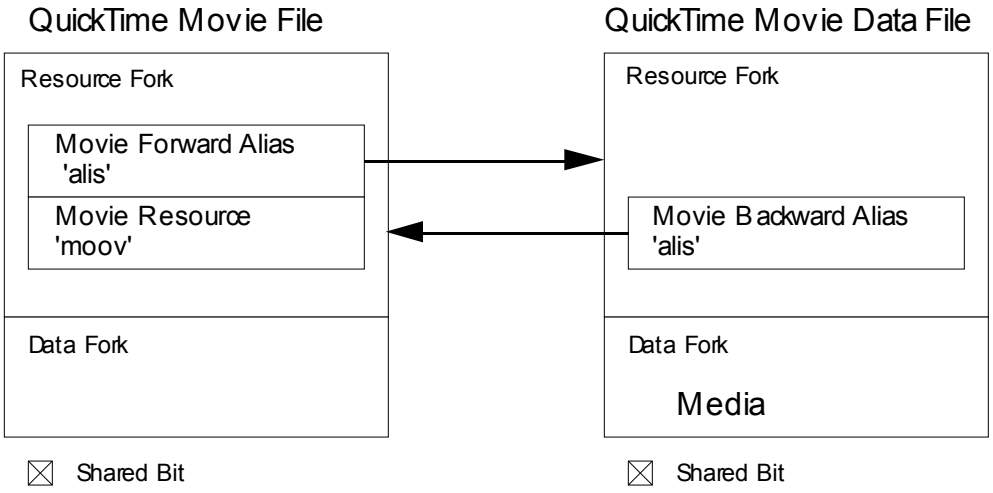

Figure 1—Diagram of Two Dependent Files

Two types of customized aliases are used: forward aliases and backward aliases. Files that reference other files contain a forward alias to the referenced file. Files that are referenced by another file contain a backward alias to the referencing file. To enable quick identification of a dependent file, the "shared" bit of the file's Finder Information is set (*Inside Macintosh* Volume VI, page 9-37).

A dependent file is a file that contains a dependent alias (either a backward or forward alias) and whose shared bit is set. If both conditions do not exist, then the file is not dependent.

New Use of Shared Bit

The shared bit is bit number 6 of the fdFldr field of the file's FInfo record. The following sample code demonstrates the proper method of identifying a dependent file.

```
OSErr FileIsDependent(FSSpec *anyFSSpec,Boolean *isShared)
{
       #define sharedBit (1<<6)
       OSErr  err;
       FInfo  fileFInfo;

       err = FSpGetFInfo(anyFSSpec,&fileFInfo);
       if (err) return err;
       if ((fileFInfo.fdFlags & sharedBit) && (fileFInfo.fdType != 'APPL') )
              *isShared = true;
       else
              *isShared = false;
       return noErr;
}
```

Previously, the shared bit applied only to applications that could be used on a network volume by multiple users. The new use does not obscure its previous use. If the file is an application and the bit is set, then it is available to multiple users and its old meaning is retained. If the file is not an application, and the bit is set then the file may depend on other files.

Dependent Alias Format

Aliases can be customized in two ways: the userType field can be modified and custom data can be added after the alias' private data structure (*Inside Macintosh* Volume VI, page 27-12).

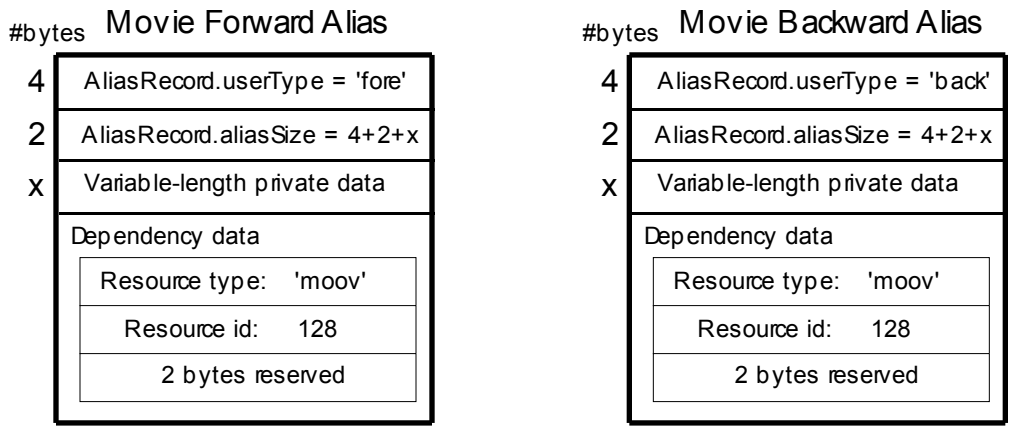

Fig. 2—Diagram: Movie Forward Alias & Movie Backward Alias From Fig. 1

For dependent aliases, the userType field of the AliasRecord contains a signature that indicates the direction of the reference. A backward alias has an alias userType of 'back'. A forward alias has an alias userType of 'fore'.

Custom data can be added to the dependent aliases to provide further identification. The aliasSize field contains the size of the alias record, and you can jump after the record to add your custom data.

QuickTime adds 8 bytes after the Alias Record. The first 8 bytes are reserved for QuickTime and describe the resource type and ID. For movies, the ResType is 'moov'. The final two bytes are reserved. The custom data format is:

```
struct {
       ResType       refResource;        // set to 0 for non-QuickTime
       short         refResourceId;      // set to 0 for non-QuickTime
       short         reserved;           // Always set to 0
       };
```

You should not alter any custom alias that your application did not create. However, you can use the information to identify and delete dependent aliases.

Working with Dependent Files

All the normal file rules apply to dependent files. Of course, you can delete them and copy them. But, in certain circumstances, you may need to operate on a referenced or referencing file’s dependent aliases and Finder Information.

Deleting Dependent Files

Deletion of a dependent file means deleting all dependent aliases, clearing the shared bit of dependent files if necessary, and finally deleting the file itself. Deletion of a file by definition means deletion of one file, and one file alone. A dependent file deletion also includes the deletion of the dependent aliases and appropriately setting the shared bit. This three-step process can be quite complex. Here are the rules:

If QuickTime is installed, use DeleteMovieFile to delete movie files. DeleteMovieFile will perform the three-step process. QuickTime uses the File Manager to do its file handling, but, in addition, it adds the logic to handle dependent files. You can also use DeleteMovieFile on non-QuickTime files without any problems.

If QuickTime is not installed or you create your own dependent files, use the File and Alias Managers to delete dependent files. The use of the File Manager and Alias Manager to delete QuickTime movie files is strongly discouraged. But, lack of QuickTime and creating your own dependent files are two situations where you will have to perform the deletion yourself.

Since aliases are part of a dependent file, you can not work with dependent files on System 6 without QuickTime. QuickTime installs the Alias Manager on System 6 and DeleteMovieFile will then be able to work with movie files.

Removing Dependent Aliases

For this discussion, target file means the file to be deleted, and the alias file means the file at the end of the dependent alias. Also, this discussion covers the deletion of a target file with a forward dependency alias, because you use the same steps for a backward dependency alias. You just need to search for a backward dependency alias instead. The seven steps to delete a dependent file are as follows:

1. Open the target file.

   For the target file you want to delete, you need to delete all dependent aliases. Thus, first you need to open the target file's resource fork. Be sure to have an FSSpec, because you will use it in step 4 below.

2. Search the target file's resource fork for forward dependent aliases.

   For each alias in the target file, you need to see if it has a forward dependent alias by checking the alias' UserType field for 'fore'. Then you need to retrieve the QuickTime custom data in order to be sure the target file is a movie file. If the custom data contains 'moov' in the ResType field, then you know you have a QuickTime dependent file.

3. Search the alias file's resource fork for backward dependent aliases.

Resolve the forward dependency alias and open the resource fork. For each alias in the file, if it is a forward alias, get the custom data from it and compare it to the custom data you stored away. If it is a 'moov' resource and the resource ID matches, resolve the alias.

4. Compare FSSpec of the target file and the resolved backward dependent alias.

   You now need to be sure you have the correct backward alias. You already have an FSSpec from the target file. You can create another FSSpec by resolving the backward alias. Call FSSpecEqual with both of these FSSpecs. If the FSSpecs are equal, you know you have the correct dependent alias.

5. Remove the backward dependent alias.

   You can now remove the alias. Be sure to update your resources correctly and close the resource file of the alias file.

6. Check the shared bit and delete the file

   As you perform these operations, be sure to keep track of other dependent aliases for the alias file. If your dependent alias was the only one, then the alias file loses its dependency and you should clear the shared bit.

7. Delete the target file.

   Finally, you need to close the resource fork of the source file and delete it.

Copying Dependent Files

Copying a dependent file means creating a new file, and a new set of dependent aliases. In addition, the shared bit should be checked to be sure it is set.

Creating Dependent Files

Many current applications create dependent files, but they do not do it uniformly. Thus, it is recommended that you format dependent files like QuickTime dependent files. It is possible future versions of system software will further exploit this information. In addition, other applications will be able to understand your files if you support this format.

To create a dependent file, you need to create your files, create the dependent aliases, and set the shared bit. To create the dependent aliases, use the normal Alias Manager routines. Set the userType field of the alias to either 'fore' or 'back'. For the custom data for the dependent aliases, you need to put zero in the first 8 bytes after the alias' private data.

Orphaned Dependent Files

Thus, the Finder without the INIT will not delete dependent files correctly. Fortunately, this inability is not terribly problematic. It means dependent aliases may be left around in files, and shared bits may be set incorrectly. These files are called *orphaned dependent files.* If your application works with dependent files, you may work with orphaned dependent files. Some dependent aliases will not be able to be resolved. Your application should be aware of this possibility and should be able to handle orphaned dependent files gracefully.

Cross-Platform Movie Files

QuickTime for Windows' movies are by definition single forked and self-contained. Therefore, dependent files do not apply to that platform.

Further Reference:

- *Inside Macintosh*, QuickTime
- *Inside Macintosh*, QuickTime Components
- QuickTime 2.0 SDK Documentation
- QT 2 - Inside Macintosh:QuickTime Components Tech Note
- QT 3 - QuickTime for Windows Tech Note