UserData

The UserData manipulation routines have been extended to allow easier access to data stored in the UserData items. Routines have also been added to allow use of UserData outside of the immediate context of QuickTime movies. Applications and other components can now create their own UserData structures. Routines are also provided to allow UserData to be stored and retrieved in a manner similar to public movies (atoms).

```
pascal OSErr SetUserDataItem(UserData ud, void *data, long size, long type, long index
        )
```

SetUserDataItem is a pointer based version of SetUserData. The size of the data pointed to must be provided, since the data may be embedded inside of a larger data structure, or may be on the stack. The data must be locked down, since SetUserDataItem may move memory. You can pass zero or one for the index to indicate the first item.

```
pascal OSErr GetUserDataItem(UserData ud, void *data, long size, long type, long index
        )
```

GetUserDataItem is a pointer based version of GetUserData. If the size field provided doesn't match the exact size of the actual UserData item, an error is returned. In this case, you would need to use GetUserData instead. GetUserDataItem is useful for retrieving small, fixed size pieces of UserData without having to create a Handle. You can pass zero or one for the index to indicate the first item.

```
pascal OSErr NewUserData(UserData *ud)
```

Creates a new UserData structure. It may be manipulated by any of the standard UserData calls. If the routine fails the “ud” field is set to nil. The only way this routine could fail is under low memory.

```
pascal OSErr DisposeUserData(UserData ud)
```

Disposes of a UserData created by NewUserData. It is OK, but silly, to pass NIL. You should only call DisposeUserData on a UserData structure which you allocated. Don't dispose of UserData references obtained from the Movie Toolbox routines GetMovie/Track/MediaUserData.

```
pascal OSErr PutUserDataIntoHandle(UserData ud, Handle h)
```

Takes the given UserData and replaces the contents of the handle with a publicly parse-able form of the UserData. The contents of “h” are appropriate for storage, much like a public movie.

```
pascal OSErr NewUserDataFromHandle(Handle h, UserData *ud)
```

Creates a new UserData structure from a handle. The handle must be in the standard UserData storage format (atoms, just like a public movie). Usually the handle will have been created by calling PutUserDataIntoHandle.

This routine should only fail from low memory. If it does, the “ud” field is set to zero.

LoadIntoRam

The LoadIntoRam calls (LoadMovie/Track/MediaIntoRam) are significantly faster in QuickTime 1.5. QuickTime 1.5 also now loads tracks into memory in a time slice order so that if the call fails because it is out of memory, you'll be left with all tracks loaded up to about the same point in time, rather than one track completely loaded, and the other only partially loaded. In addition a number of flags have been defined (if you'll recall, in 1.1 the flag parameter came with instructions to always set to zero). These flags give you explicit control over what is loaded into memory, and how long it is kept around.

The LoadIntoRam flags can be set in any combination which seems to make sense to you. This probably means that setting the "keepInRam" and "flushFromRam" flags might not be ideal.

```
enum {
            keepInRam = 1<<0,
            unkeepInRam = 1<<1,
            flushFromRam = 1<<2,
            loadForwardTrackEdits = 1<<3,
            loadBackwardTrackEdits = 1<<4
}
```

`keepInRam`

> All data loaded with this flag set will be made non-purgable. This means that it will not be released from memory until you explicitly request it. This can fill up your heap very quickly. Be careful.

`unkeepInRam`

> This makes all indicated data purgable. It is not necessarily released from memory immediately. Currently, whether or not a chunk can be purged is maintained internally by a single bit. This means there is no counter. Therefore, if you care very very much about things you have to work very very hard and walk the edit list a lot.

`flushFromRam`

> All indicated data is purged from memory, unless it is currently in use by a media handler (for example, it is still drawing frames from the requested times). This is a much more vicious version of "unkeepInRam" - it makes the memory available for purging, and then performs the purge. You may want to use this option if you are particularly tight on memory.

`loadForwardTrackEdits`

`loadBackwardTrackEdits`

> In some cases, an edited movie plays back much more smoothly if the data around edits are already in RAM. By setting either of these flags you can load only the data around edits - the movie toolbox walks the edits and decides the right amount

of data to load for you. If you are going to play the movie forward only set “loadForwardTrackEdits”; if you are going to only play the movie backwards, set “loadBackwardTrackEdits “. If you are going to play in both directions, or you don’t know, set both flags.

High Level Editing

QuickTime 1.5 provides support for pasting different data types into a movie. For example, QuickDraw Pictures and standard sound data, can be pasted directly into a movie. If you are using the Movie Controller, this support happens transparently. You don’t need to change anything in your application. If you are using calling the Movie Toolbox directly to do editing, some new support calls are provided for pasting in any data type.

```
pascal Component IsScrapMovie(Track targetTrack)
```

This routine looks on the system scrap to see if it could translate any of the data on the scrap into a form that could be used in a movie. If it finds an appropriate type, it returns a MovieImport Component which could translate the scrap. Otherwise it returns 0.

As an option you can pass in a particular track which you are interested in adding data to. In this case, a MovieImport Component will only be returned if it can convert the data on the scrap to a format appropriate to the given track.

```
pascal OSErr PasteHandleIntoMovie(Handle h, OSType handleType, Movie theMovie, long
                flags, ComponentInstance userComp)
```

PasteHandleIntoMovie takes the contents of the given handle, together with its type, and pastes it into the given movie. If the handle is set to zero, the scrap is searched for a field of the type “handleType”. If both “h” and “handleType” are nil, then the first available data from the scrap is used.

If you are just pasting in data from the scrap, it is best to allow PasteHandleIntoMovie to retrieve the data from the scrap, rather than doing it yourself. In this way, it will be able to obtain supplemental data from the scrap if necessary (i.e. ‘styl’ resources for ‘TEXT’).

The only currently defined flag is “pasteInParallel” which turns the “Paste” call into an “add” call. However, the add may not necessarily create a new track, but rather use a piece of an existing track if possible.

PasteHandleIntoMovie always pastes into the current selection using the following rules.
1. If the selection is empty (i.e. duration == 0), it simply adds the data with whatever duration seems appropriate.
2. If the selection is not empty, the data is added and then scaled to fit into the duration of the selection. The current selection is deleted, unless you set the “pasteInParallel” flag.

If you want a particular MovieImport Component to perform the conversion, you may pass the Component or an instance of that Component in the “userComp” field. Otherwise set it to zero to allow the Movie Toolbox to choose the appropriate Component to use. If you pass in a ComponentInstance it will be used by PasteHandleIntoMovie. This allows you to communicate directly with the Component before making this call to establish any conversion parameters.

```
pascal OSErr PutMovieIntoTypedHandle(Movie theMovie, Track targetTrack, OSType type,
        Handle handleData, TimeValue start, TimeValue dur, long flags,
        ComponentInstance userComp)
```

This routine takes a movie, and optionally just a single track from within that movie, and converts it into a handle of a given type. You specify the movie to convert in “theMovie” and a particular track, if any, in the “targetTrack” parameter. The type of the new data should be given in the “type” field and the actual handle to put the data into is the “handleData” field. The segment of the movie to be converted should be specified in the “start” and “dur” parameter. Set the “flags” to 0.

If you want a particular MovieExport Component to perform the conversion, you may pass the Component or an instance of that Component in the “userComp” field. Otherwise set it to zero to allow the Movie Toolbox to choose the appropriate Component to use. If you pass in a ComponentInstance it will be used by PutMovieIntoTypedHandle. This allows you to communicate directly with the Component before making this call to establish any conversion parameters.

File Conversions

```
pascal OSErr ConvertFileToMovieFile(const FSSpec *inputFile, const FSSpec *outputFile,
        OSType creator, ScriptCode scriptTag, short *resID, long flags,
        ComponentInstance userComp, MovieProgressProcPtr proc, long refCon)
```

ConvertFileToMovieFile takes the given “inputFile” and converts it to a movie file in the “outputFile.” If the output file is a new file, you can supply a “creator” for the file. The “scriptTag”, “resID”, and “flags” parameters are all the same as in CreateMovieFile.

Because some conversions may take a non-trivial amount of time you can pass a standard movie progress procedure in the “proc” and “refCon” fields.

If you want a particular MovieImport Component to perform the conversion, you may pass the Component or an instance of that Component in the “userComp” field. Otherwise set it to zero to allow the Movie Toolbox to choose the appropriate Component to use. If you pass in a ComponentInstance it will be used by ConvertFileToMovieFile. This allows you to communicate directly with the Component before making this call to establish any conversion parameters.

```
pascal OSErr ConvertMovieToFile(Movie theMovie, Track onlyTrack, const FSSpec
        *outputFile, OSType fileType, OSType creator, ScriptCode scriptTag, short
        *resID, long flags, ComponentInstance userComp)
```

ConvertMovieToFile takes the given movie, and optionally just a single track within that movie, and converts it into an file specifed by “outputFile” having the type of “fileType”. If the output file is a new file, you can pass the “creator” and “scriptTag” fields. Set the “flags” to zero.

If you want a particular MovieExport Component to perform the conversion, you may pass the Component or an instance of that Component in the “userComp” field. Otherwise set it to zero to allow the Movie Toolbox to choose the appropriate Component to use. If you pass in a ComponentInstance it will be used by ConvertMovieToFile. This allows you to communicate directly with the Component before making this call to establish any conversion parameters.

Data Fork Movies

For QuickTime 1.5, two new routines are provided for storing and retrieving movies stored in the data fork of a file. These routines provide more robust data reference resolution and better low memory performance than was possible using NewMovieFromHandle and PutMovieIntoHandle in QuickTime 1.0.

```
pascal OSErr NewMovieFromDataFork( Movie *theMovie, short fRefNum, long fileOffset,
            short flags, Boolean *dataRefWasChanged)
```

NewMovieFromDataFork lets you retrieve a movie stored anywhere in the data fork of a file. You pass in a pointer to the movie to be created. The file must already be open, and you pass in the file reference in the “fRefNum” parameter. Pass in the starting file offset of the public movie in the “fileOffset” parameter. The “flags” field contains the standard “newMovie” flags. The “dataRefWasChanged” parameter is the same as for all new movie calls, and may be set to zero.

```
pascal OSErr PutMovieIntoDataFork( Movie theMovie, short fRefNum, long offset, long
            maxSize)
```

PutMovieIntoDataFork lets you store a public movie version of “theMovie” in the data fork of a file. You pass in an open write path in the “fRefNum” parameter. The “offset” parameter indicates where the movie should be written. The “maxSize” parameter indicates the largest number of bytes that may be written. If necessary the file will be extended. If there is insufficient space to write the movie, either due to a lack of disk space, or because of the limit given in “maxSize”, a “dskFullErr” will be returned. If there is no limit on how much space the movie may take up in the file, pass 0 for “maxSize”.

Miscellaneous

```
pascal Fixed GetTrackEditRate(Track t, TimeValue atTime)
```

Returns the rate of the track edit of the given track at the indicated time. If an invalid time or track is passed, the returned value is 0.0. The track edit rate is typically 1.0, unless ScaleMovie/TrackSegment has been called. This call is really only interesting if you are parsing movie data directly in your application, or you are a client of the Generic Media Handler.

```
pascal OSErr SetMediaDataRef( Media media, short index, Handle blob, OSType blobType)
```

Don’t call this routine unless you have a really good reason.

SetMediaDataRef let’s you change which file the given media thinks its data is stored in. You pass it a data reference, and the data reference type. Typically this will be a file alias and ‘alis’. As with all data reference calls the index starts with 1. Making this call can destroy your world. However, if you want to resolve you own missing data references, or are developing a special purpose kind of application, this call may be really useful.

```
pascal OSErr SetMediaSampleDescription( Media theMedia, long index,
            SampleDescriptionHandle descH)
```

This is one of the most dangerous calls you will ever make. You should probably only ever make it on an inactive track. It lets you change the contents of a particular SampleDescription of a given Media. This can be useful in the case of a Media Handler, such as text, that stores playback information in its SampleDescription, as opposed to just data format information as in the case of the Video Media Handler.

The index must be between 1 and the largest sample description index. Do not make this call unless you are sure of what you are doing.

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

This routine winds its way up a chain of TimeBases until it reaches a Master Clock. Along the way it multiplies together the rates of all the TimeBases it encounters. In this way it can return to you the effective rate that the given TimeBase is moving at, relative to the Master Clock to which it is slaved.

This routine is useful when you need to make scheduling decisions based on a TimeBase's rate, for example when writing media handler. By calling GetTimeBaseEffectiveRate rather than GetTimeBaseRate you can easily take into effect any TimeBase slaving that may be in effect.

```
callBackAtExtremes
```

A new callback type has been added, callBackAtExtremes (better name suggestions welcome). This callback type calls you back either when the TimeBase reaches it start time, its stop time, or either. If the start or stop time of the TimeBase 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 will fire with the flags “triggerAtStart” and “triggerAtStop”. The flags may both be set.

The Clock Component and related CallBack interface have been enhanced slightly to support this type of CallBack.