Media Handler Open/Close/Target

Writing a media handler that relies on the standard media handler means that you have to do a few special things in your general component code.

In its open code, your media handler must open an instance of the Generic Media Handler. It must also tell the Generic Media Handler that it is owned by your media handler by sending the Component Manager SetTarget message. You must keep track of the Generic Media Handler instance in your globals so you can call it directly and delegate calls to it.

In your close code, you must close the Generic Media Handler you opened.

You must implement the Component Manager ComponentSetTarget routine so you know if you are being used by another media handler.

Your dispatcher must delegate any routine selectors it doesn't understand or implement to the Generic Media Handler. This is done using DelegateComponentCall.

Sample code for all of this is given at the end of this document.

Note

After each call description there are two subsections: "when to implement" and "flag dependencies".

The "when to implement" section attempts to explain under what circumstances you would need to implement that particular routine. Most routines are optional.

The "flag dependencies" section tries to explain which handler capability flags have an impact on how/if a particular call is used. Handler capability flags are explained below.

Media Handler Management

```
pascal ComponentResult MediaInitialize (ComponentInstance ci, GetMovieCompleteParams *gmc)
```

After your media has been opened, it receives a MediaInitialize call. It is passed a structure containing tons of useful pieces of information about the Media it is attached to. You should make copies of any fields that you need from this structure. You do not need to dispose of any of the structures you copy from this record (i.e. the matte PixMap). They are owned by the Movie Toolbox.

At the time MediaInitialize is called, the complete movie has been loaded from the public movie. If the media is not being created from a public movie, but rather by an application (or editing), MediaInitialize is still called.

You may receive other calls in between your Open and MediaInitialize. In particular, MediaGetMediaInfo and MediaPutMediaInfo may be called.

If you return an error from MediaInitialize and the movie is being loaded in from disk, the track will be permanently disabled. If you return an error when the media is initially being created, the media will be disposed of immediately.

Inside your MediaInitialize routine should call MediaSetHandlerCapabilities (described next) to inform the Generic Media Handler of what services your Media Handler will require.

When to implement: All media handlers will probably want to implement this call so that they can remember certain pieces of information, such as the track they are associated with, or the track dimensions. If you cache any values that may change (track dimensions, matrix, rate, etc.) you should make sure to implement the appropriate “MediaSet” routines described below, so you will be notified when these values change.

Flag dependencies: None.

```
pascal ComponentResult MediaSetHandlerCapabilities (ComponentInstance ci, long flags, long
        flagsMask)
```

Your media handler doesn’t implement this routine, but rather makes this call to the Generic Media Handler.

You can make this call at any time, but typically it is made once from the your MediaInitialize call. Pass the flags and a mask to indicate which flags you are effecting (same idea as SetMoviePlayHints).

By default all flags are false, so if you don’t make this call, your media handler won’t be able to do much except store and retrieve data.

```
enum {
      handlerHasSpatial = 1<<0,
      handlerCanClip = 1<<1,
      handlerCanMatte = 1<<2,
      handlerCanTransferMode = 1<<3,
      handlerNeedsBuffer = 1<<4,
      handlerNoIdle = 1<<5,
      handlerNoScheduler = 1<<6,
      handlerWantsTime = 1<<7,
      handlerCGrafPortOnly = 1<<8
}
```

handlerHasSpatial - Your media handler want room on the screen to draw, and wants to receive calls relating to spatial changes.

handlerCanClip -Your media handler can handle clipping on its own. You will receive MediaSetClip calls.

handlerCanMatte - Your media handler can handle matting on its own. More later. Don’t set this bit.

handlerCanTransferMode - Your media handler can apply transfer modes other than srcCopy/ditherCopy on its own.

handlerNeedsBuffer - To force your drawing to go into an offscreen buffer. The Generic Media Handler takes care of managing the offscreen buffer for you.

handlerNoIdle - You don’t need MediaIdle calls. Only useful for media that act as a data store, rather than actually drawing/playing/controller whatever.

handlerNoScheduler - You do hard work in your MediaIdle call. Otherwise you only get MediaIdle calls when you have to draw.

handlerWantsTime - Set this flag if your media handler needs to be called with time at a time other than when a sample changes (or a sample must be drawn). If the flag is set your

MediaIdle routine will be called as often as possible, but you have a new sample when the "mMustDraw" flag is set.

handlerCGrafPortOnly - Set this flag if your media handler cannot deal with being passed a Classic QuickDraw GrafPort (a non-CGrafPtr). If this flag is set, the Generic Media

Handle will force drawing into a Color GrafPort. This flag is only necessary if your Media Handler relies on having a CGrafPort available for drawing.

When to implement: Your media handler should never implement this call.

Flag dependencies: None.

```
pascal ComponentResult MediaGGetStatus (ComponentInstance ci, ComponentResult *statusErr)
```

Your media is called for status when the toolbox calls GetMovieStatus or GetTrackStatus are made. If you are having problems playing, you should put an error into statusErr. Do not return the error as your result code.

When to implement: Your media handler should implement this call if it may encounter situations where it is unable to do its work at idle time. The reasons may include low memory, no sound channels, or missing hardware features. If you don't implement this call, it is assumed that your status error is always 0.

Flag dependencies: None.

Media Task

```
pascal ComponentResult MediaIdle (ComponentInstance ci, TimeValue atMediaTime, long flagsI
            long *flagsOut)
```

If your media does any work at idle time, here's where it happens. You are given the current media time to display a sample at.

If you are letting the Generic Media Handler do your scheduling for you, this is an easy call. All you have to do is call GetMediaSample to get the sample to draw, do your drawing, and get out.

If you are doing your own scheduling (you set the handlerNoScheduler flag) things are a bit more difficult. You need to look at flagsIn. If "mPreflightDraw" is set, then you shouldn't draw on this call. You should just see if you need to draw and return "mNeedsToDraw" in the flagsOut. If the "mMustDraw" flag is set, you have to draw. If neither of these flags is set, it is up to you to decide whether or not you need to draw. However, if you do draw, you must be sure to set the 'mDidDraw" flag in flagsOut.

Be sure not to change the GWorld on exit. If you change any values in the port, be sure to change them back. Be aware that you could be drawing into an old style GrafPort.

This call is also used for putting movie frames into a picture. Therefore, if you are being totally slimy and not using QuickDraw for your drawing, you should look at the port's picSave field and react accordingly.

When to implement: If your media handler needs time to do any work at all while the movie is playing, you must implement this routine.

Flag dependencies: Not called if handlerNoIdle is set.

```
pascal ComponentResult MediaPreroll (ComponentInstance ci, TimeValue movieTime, Fixed rate
```

This call is made to your media handler when the application prerolls the movie. At this time, your media handler should perform any necessary set up for playback starting at the given “movieTime”, playing at the given “rate”.

When to implement: If your media handler performs any non-trivial setup in order to play, you should implement this routine to prevent stutters when starting the movie.

Flag dependencies:

Load & Save / Copy & Paste

```
pascal ComponentResult MediaPutMediaInfo (ComponentInstance ci, Handle h)
```

If your media maintains information beyond its samples and the information stored by the Generic Media Handler (transfer mode, op color, and sound balance), you can use this call to store it. When you receive this call, put whatever information you wish to store into the handle provided. You must resize the handle as necessary.

It is a good idea to keep a version number in your MediaInfo, for future changes.

Do not dispose of the handle passed to you.

When to implement: Your media handler should implement this call if you need to store data beyond what is already stored automatically for you. If you don’t implement this call, no additional data is stored.

Flag dependencies: None.

```
pascal ComponentResult MediaGetMediaInfo (ComponentInstance ci, Handle h)
```

If additional information has been stored with your media using the MediaPutMediaInfo call, the you will receive a MediaGetMediaInfo call when the media is recreated (from disk or by editing). You will be passed a handle of data, the same data that was stored with MediaPutMediaInfo.

This call may occur either before or after MediaInitialize depending on whether the media is being created from a public movie or during an edit operation.

Do not dispose of the handle passed to you.

When to implement: If you expect to receive additional data stored with your media, you should implement this call. If you don’t implement this call, you will not receive the additional media information.

Flag dependencies: None.

General Media State

```
pascal ComponentResult MediaSetActive (ComponentInstance ci, Boolean enableMedia)
```

Your media receives this call when the active state of the media changes. When you media is deactivated, you might dispose of any temporary storage you were using to play.

Your initial enabled state is always disabled (false).

When to implement: If you are doing your own scheduling (you've set the handlerNoScheduler flag) you probably want to know if the media is enabled or not. You may also want to implement this call if you allocate a significant amount of storage (or other resources) in order to play.

Flag dependencies: None.

```
pascal ComponentResult MediaSetRate (ComponentInstance ci, Fixed rate)
```

When the rate of the movie changes, your MediaSetRate is called. The rate passed is the new rate. The rate is actually the movie's effective rate (details details).

When to implement: If you are doing your own scheduling (you've set the handlerNoScheduler flag) you probably want to know if the media is playing and/or in what direction.

Flag dependencies: None.

```
pascal ComponentResult MediaTrackEdited (ComponentInstance ci)
```

When the user performs an edit on the media, your MediaTrackEditied routine is called. If you had stored the locations of any track edits, or have any stored movie time values, you should invalidate them here.

When to implement: If you are storing values in movie time, you should implement this call so you know when to invalidate them.

Flag dependencies: None.

```
pascal ComponentResult MediaSetMediaTimeScale (ComponentInstance ci, TimeScale newTimeScal
```

If the media time scale is changed, this routine is called with the new value.

When to implement: If you store media time values, you need to implement this call to know when to invalidate/update them.

Flag dependencies: None.

```
pascal ComponentResult MediaSetMovieTimeScale (ComponentInstance ci, TimeScale newTimeScal
```

If the movie's time scale is changed, this routine is called with the new value.

When to implement: If you store movie time values, you need to implement this call to know when to invalidate/update them.

Flag dependencies: None.

Media Graphics State

```
pascal ComponentResult MediaSetGWorld (ComponentInstance ci, CGrafPtr aPort, GDHandle aGD)
```

If you need to know when the GWorld into which you are drawing changes, you should implement this call. The new CGrafPort and GDevice are passed in.

When you idle routine is called, the correct port to draw into is already set, so you only really need to implement this routine if you are going to do specialized graphics stuff.

When to implement: Only if you care about if you are drawing a GrafPort vs. a CGrafPort or are using an Image Compression Manager decompression sequence.

Flag dependencies: Not called unless handlerHasSpatial is set.

```
pascal ComponentResult MediaSetDimensions (ComponentInstance ci, Fixed width, Fixed height
```

When the call SetTrackDimensions is made, your MediaSetDimensions routine is called to inform you of the new track width and height.

When to implement: If your media handler does spatial stuff, you want to implement this routine to know when the track dimensions are changed.

Flag dependencies: Not called unless handlerHasSpatial is set.

```
pascal ComponentResult MediaSetMatrix (ComponentInstance ci, MatrixRecord *trackMovieMatri
```

If the movie or track matrices are changed, your MediaSetMatrix routine is called with the new display matrix. The matrix passed in is the track matrix concatenated with the movie matrix. Therefore passing the track box (defined by {0, 0, trackWidth, trackHeight} through this matrix, you can determine the display bounds.

You should call GetMatrixType and make sure that you media handler can deal with the matrix provided. If you can’t, you should call MediaSetHandlerCapabilities and set the handlerNeedsBuffer flag to force your drawing to go into an offscreen.

When to implement: If your media handler does spatial stuff, you want to implement this routine to know when the display matrix has changed.

Flag dependencies: Not called unless handlerHasSpatial is set.

```
pascal ComponentResult MediaSetClip (ComponentInstance ci, RgnHandle theClip)
```

If the clip for your track changes, MediaSetClip is called. You are passed in a new clip region. You are responsible for disposing of the region provided. The given clip is in display space, and includes the effects of both the track source clip and any compositing.

Flag dependencies: Not called unless handlerHasSpatial and handlerCanClip are set.

```
pascal ComponentResult MediaGetTrackOpaque (ComponentInstance ci, Boolean *trackIsOpaque)
```

This call is made by the Movie Toolbox when it is trying to figure out the movie's compositing. If your media does any drawing that depends on what is drawn behind it (i.e. draws in blend mode), you should set this flag to true. Otherwise you should leave the flag alone.

When to implement: If your media handler does really cool semi-transparent bouncing shapes, you will want to implement this routine. Or if you handle transfer modes on your own (have set the handlerCanTransferMode flag).

Flag dependencies: Not called unless handlerHasSpatial is set.

```
pascal ComponentResult MediaGetNextBoundsChange (ComponentInstance ci, TimeValue *when)
```

The Movie Toolbox makes this call to you when it is trying to figure out the next spatial change in the movie. The current time should be used as a starting point. The current rate should be used as the direction (with 0 implying forward). The TimeValue returned should be in Movie time, not media time.

When to implement: If your media handler changes the shape of the track over time, you need to implement this routine.

Flag dependencies: Not called unless handlerHasSpatial is set.

```
pascal ComponentResult MediaGetSrcRgn (ComponentInstance ci, RgnHandle rgn)
```

When calculating the shapes and clips of the movie's track, the Movie Toolbox calls this routine to give the media handler a chance to define the actual area of the track box that it is using. This allows tracks to be non-rectangular, or to only use a portion of their track box. The region provided is already allocated and initialized to the rectangle defined by the track dimensions. You may change the region as necessary. Do no dispose it. The returned region should be in track source space, not movie or display space.

When to implement: If your media handler doesn't completely fill the track box it is given, you should implement this routine so tracks behind it can draw in that area.

Flag dependencies: Not called unless handlerHasSpatial is set.

Media Sound State

```
pascal ComponentResult MediaGSetVolume (ComponentInstance ci, short volume)
```

When the volume of your track is changed, MediaGSetVolume is called with the new volume. The volume is an 8.8 fixed point number (just like the Movie or Track volume). The volume passed is the track volume scaled by the movie volume. It has not been scaled by the Macintosh speaker volume (which is what you want if you are playing through the Sound Manager).

When to implement: If your media handler plays sounds, your should implement this call so that you are notified when the volume changes.

Flag dependencies: None.

```
pascal ComponentResult MediaSetGraphicsMode (ComponentInstance ci, long mode, RGBColor
            *opColor)
```

Don’t implement this yet.

```
pascal ComponentResult MediaGetGraphicsMode (ComponentInstance ci, long *mode, RGBColor
            *opColor)
```

Don’t implement this yet.

```
pascal ComponentResult MediaSetSoundBalance (ComponentInstance ci, short balance)
```

Don’t implement this yet.

```
pascal ComponentResult MediaGetSoundBalance (ComponentInstance ci, short *balance)
```

Don’t implement this yet.

Sample Media Handler Routines

```
pascal ComponentResult ComponentTarget(ComponentInstance c, ComponentInstance parentCompon
            = ComponentCallNow(kComponentTargetSelect,4);


/*
            Component Dispatcher
*/
pascal ComponentResult MyMediaHandler( ComponentParameters *params, Handle storage )
{
            ComponentRoutine theRtn = 0;

            switch (params->what)
            {
                  case kComponentOpenSelect:    theRtn = (ComponentRoutine) MyMediaOpen;
                  case kComponentCloseSelect:   theRtn = (ComponentRoutine) MyMediaClose;
                  case kComponentVersionSelect: return 0x00010000;
                  case kComponentTargetSelect:  theRtn = (ComponentRoutine) MyMediaTarget;
            }

            if (theRtn)
                  return CallComponentFunctionWithStorage(storage, params, theRtn);
            else {
                  ComponentInstance delegateComponent;
                  if (storage && (delegateComponent = (**(MyMediaGlobals)storage).delegate
                        return DelegateComponentCall(params, delegateComponent );
                  else
                        return noErr;
            }
}
```

```
/*
            Component call Open handler
*/
pascal ComponentResult MyMediaOpen(MyMediaGlobals globals, ComponentInstance self)
{
            OSErr err = noErr;
            ComponentInstance delegateComponent;

            globals = (MyMediaGlobals)NewHandleClear(sizeof(MyMediaGlobalsStruct));
            if (err = MemError()) return err;

            SetComponentInstanceStorage(self, (Handle)globals);
            (**globals).self = self;
            (**globals).parent = self;

            delegateComponent = OpenDefaultComponent('mhlr','gnrc');
            (**globals).delegateComponent = delegateComponent;

            if (delegateComponent)
                  ComponentSetTarget(self, self);
            else {
                  SetComponentInstanceStorage(self, nil);
                  DisposHandle((Handle)globals);
                  err = cantOpenHandler;
            }

            return err;
}

/*
            Component call Close handler
*/
pascal ComponentResult MyMediaClose(MyMediaGlobals globals, ComponentInstance self)
{
            if (globals) {
                  if ((**globals).delegateComponent)
                        CloseComponent((**globals).delegateComponent);

                  DisposHandle((Handle) globals);
            }

            return noErr;
}

/*
            Component call Target handler
*/
pascal ComponentResult MyMediaTarget(MyMediaGlobals globals, ComponentInstance parentCompo
{
            (**globals).parent = parentComponent;
            ComponentTarget((**globals).delegateComponent, parentComponent);

            return noErr;
}
```