# QuickTime Component Creation Guide

# Contents

# Listings

# Introduction to QuickTime Component Creation Guide

This book tells you how to build new components to extend the capabilities of QuickTime. The component types covered by this book include:

■ **Media handler** components, which allow the Movie Toolbox to manipulate the data in a media. These media handlers isolate the Movie Toolbox (and the applications programmer) from the details of how and where a media is stored. They are also called **derived** media handlers because they are derived from a base media handler, provided by Apple. The base media handler component handles most of the duties that are common to all media handlers, freeing the component developer to focus on the task of reading and writing a particular media type.

■ **Preview** components, which create or display a preview of a QuickTime movie file. The preview is typically displayed as part of an Open File dialog; it is normally an image, but it may contain text, sound, or other data. The preview may be contained in the movie file or it may be created on the fly by the preview component whenever it is needed.

> **Note:** This book replaces two previously separate Apple documents: "Media Handlers: Creating Media Handler Components" and "Preview Components."

In general, only developers who are creating a new media handler or preview component need to read this book.

## Organization of This Document

This book contains the following chapters:

■ About QuickTime Media Handler Components (page 9) describes what media handler components are and how they are used.

■ Creating a Derived Media Handler Component (page 13) describes the process of creating a derived media handler component.

■ Derived Media Handler Support (page 19) defines the functions you must support if you are creating a derived media handler, the functions that you may optionally support, and utility functions available to your component from the base media handler.

■ Creating Preview Components (page 31) describes how to create your own preview component. A listing of a sample component is included.

■ Functions For Displaying Previews (page 35) describes the functions for displaying previews, handling events in previews, and creating previews that are provided by preview components.

# See Also

For general information about media handler components, see *QuickTime Media Types and Media Handlers Guide*. This book introduces the idea of QuickTime media handler components and provides details of the video, sound, text, timecode, and tween media handlers.

For general information about preview components, see *QuickTime Movie Internals Guide*. This book also covers some of the technology present inside QuickTime movies, including time management, modifier tracks, access keys, and movie posters.

Information about creating more types of QuickTime components (other than those covered in this book) is included in books about those components. See the following:

■ Information about creating data handler components is in *QuickTime Transport and Delivery Guide*.

■ Information about creating movie data exchange components is in *QuickTime Import and Export Guide*.

■ Information about creating image transcoder components is in *QuickTime Compression and Decompression Guide*.

■ Information about creating video effect components is in *QuickTime Video Effects and Transitions Guide*.

■ Information about creating tween components is in *QuickTime Media Types and Media Handlers Guide*.

■ Information about creating video digitizer components is in *QuickTime Movie Creation Guide*.

■ Information about creating video output components is in *QuickTime Transport and Delivery Guide*.

The following additional Apple books cover related aspects of QuickTime programming:

■ *QuickTime Overview* gives you the starting information you need to do QuickTime programming.

■ *QuickTime Movie Basics* introduces you to some of the basic concepts you need to understand when working with QuickTime movies.

■ *QuickTime Guide for Windows* provides information specific to programming for QuickTime on the Windows platform.

# About QuickTime Media Handler Components

This chapter provides background information about media handler components in general and explains the difference between media handler components and derived media handler components. After reading this chapter you should understand why media handler components exist and whether you need to create a derived media handler component.

## Media Handler Components

Media handler components allow the Movie Toolbox to play a movie's data. The Movie Toolbox, by itself, cannot read or write movie data. Rather, media handlers perform input and output services on behalf of the Movie Toolbox. The Movie Toolbox gains access to the appropriate media handler for a particular movie track by examining the track's media. That data structure identifies the media handler that created and maintains the media.

Each media handler is primarily responsible for understanding the format and content of the media type it supports. The media handler is intimately familiar with the sample structure used in its media, the compression techniques used to store the media's sample data, and the performance characteristics of the device that stores the media.

During movie playback, the media handler draws its media's data on the screen and plays the media's sounds. The media handler may use the services of other managers such as the Image Compression Manager for compressed image data and the Sound Manager for sound data. When an application creates a movie, media handlers store the movie's data. The actual reading and writing of media data are performed by another component, the **data handler.**

Applications never directly use the services of media handlers. The Movie Toolbox controls all movie data storage and retrieval on behalf of QuickTime applications.

Figure 1-1 shows the logical relationships between applications, the Movie Toolbox, media handlers, and data handlers.
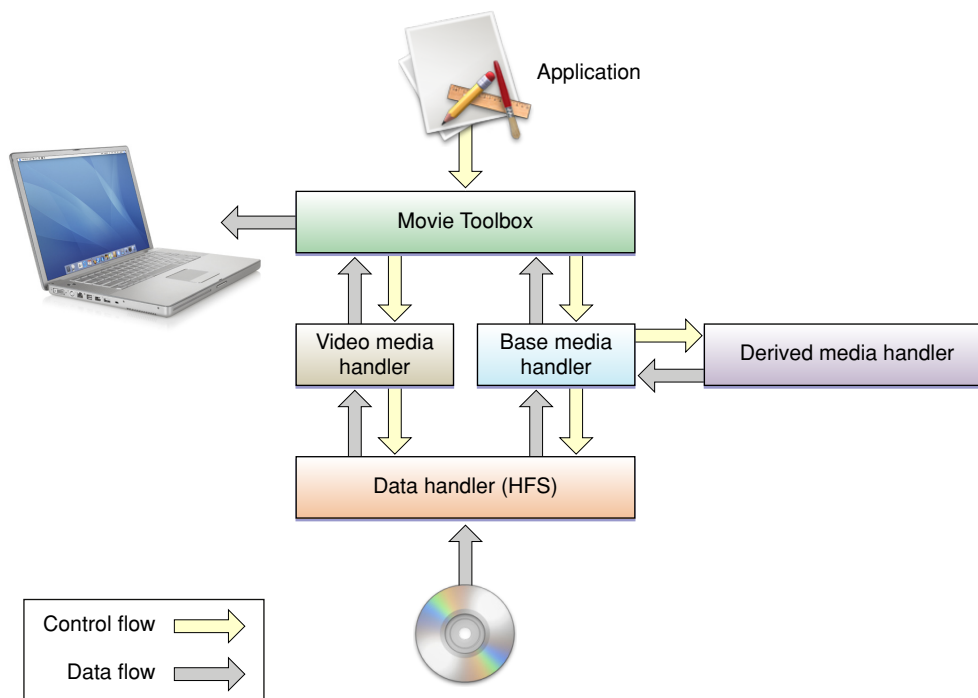
Apple had three primary goals for isolating the Movie Toolbox and QuickTime applications from the details of media data access. First, the isolation allows programmers who develop the Movie Toolbox and QuickTime applications to focus on the specifics of the problems they are addressing, freed from concerns about data access. Second, this architecture allows QuickTime to be easily extended to accommodate new storage devices and technologies. Third, by documenting the media handler interface, developers can create their own, special-purpose media handlers that work with QuickTime.

# Derived Media Handler Components

Much of what a media handler component must do is common to all media handlers. Managing a connection with the appropriate data handler, retrieving movie data from media samples, and storing movie data into new samples account for a substantial part of every media handler's responsibilities. To make it easier for developers to create media handler components, Apple provides a base media handler component that performs most of the common duties of a media handler.

Apple's base media handler component eliminates much of the work you would have to do to create your own media handler component. The base media handler interacts with both the Movie Toolbox and the appropriate data handler, so that your media handler only has to deal with service requests, and you can ignore many of the housekeeping functions. It understands the format of Apple's media samples and sample descriptions, so that your media handler only has to worry about the actual media data. Finally, it provides basic services that your media handler can use to accommodate unusual display environments.

When you build your media handler component on top of the base media handler, your media handler is known as a *derived media handler component*. This terminology is borrowed from object-oriented development and refers to the fact that your media handler is based on, or derived from, the services provided by Apple's

base media handler. All media handlers written by developers are derived media handlers, because they all use the services of the base media handler. Figure 1-2 shows the relationship between the base media handler, derived media handlers, the Movie Toolbox, and data handler components.
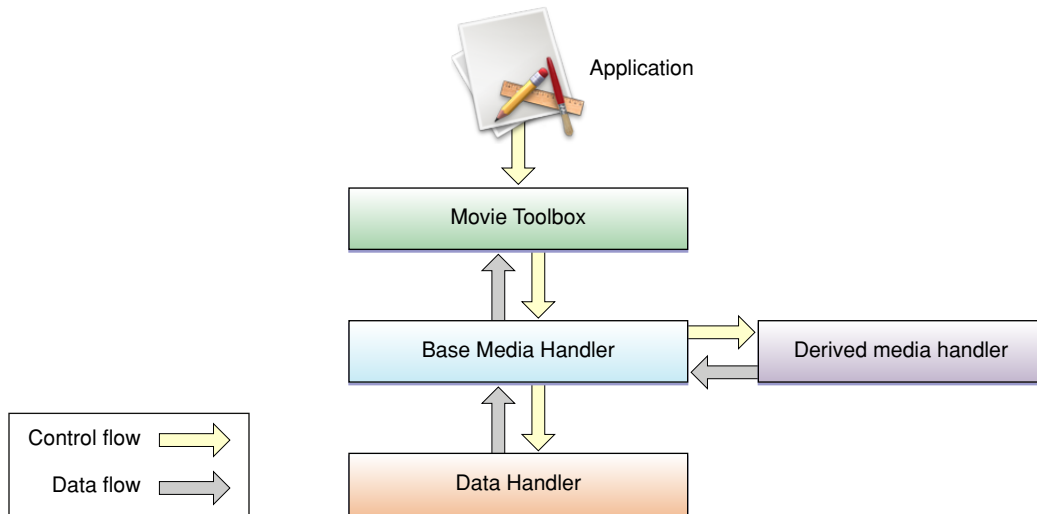


> **Note:** Early versions of QuickTime required some developers to create complete stand-alone media handlers for performance reasons; derived media handlers were limited to low-bandwidth media. This is no longer true. To create a new media handler for QuickTime, you always write a derived media handler.

# Creating a Derived Media Handler Component

This chapter describes the process of creating a derived media handler component. It provides an example of creating a derived media handler component and defines its component and request flags. The functional interface that your derived media handler component must support is described in Derived Media Handler Support (page 19).

Before reading this section, you should be familiar with how to create components.

Apple has defined the `MediaHandlerType` component type value (`'mhlr'`) for media handler components. All components of this type have the same type value.

Apple has also defined a functional interface for derived media handler components. For information about the functions that your component must support, see Derived Media Handler Support (page 19).

## A Sample Derived Media Handler Component

This section supplies a sample program that implements a derived media handler component for PICT images.

### Implementing the Required Component Functions

Listing 2-1 supplies the component dispatchers for the media handler component for PICT images together with the required functions.

**Listing 2-1**      Implementing the required media handler component functions

```
typedef struct {
    ComponentInstance       self;
    ComponentInstance       parent;
    ComponentInstance       delegateComponent;
    Fixed                   width;
    Fixed                   height;
    MatrixRecord            matrix;
    Media                   media;
    Track                   track;
} PictGlobalsRecord, *PictGlobals;

pascal ComponentResult PictMediaDispatch (ComponentParameters *params,
                                          Handle storage)
{
    OSErr err = badComponentSelector;
    ComponentFunction componentProc = 0;

    switch (params->what) {
        case kComponentOpenSelect:
            componentProc = PictOpen; break;
```

```
        case kComponentCloseSelect:
            componentProc = PictClose; break;
        case kComponentCanDoSelect:
            componentProc = PictCanDo; break;
        case kComponentVersionSelect:
            componentProc = PictVersion; break;
        case kComponentTargetSelect:
            componentProc = PictVersion; break;
        case kMediaInitializeSelect:
            componentProc = PictInitialize; break;
        case kMediaIdleSelect:
            componentProc = PictIdle; break;
        case kMediaSetDimensionsSelect:
            componentProc = PictSetDimensions; break;
        case kMediaSetMatrixSelect:
            componentProc = PictSetMatrix; break;
    }
    if (componentProc)
        err = CallComponentFunctionWithStorage (storage, params,
                                                    componentProc);
    else
        err = DelegateComponentCall (params, ((PictGlobals)
                                        storage)->delegateComponent);
    return err;
}

pascal ComponentResult PictCanDo (PictGlobals globals,
                                        short ftnNumber)
{
    switch (ftnNumber) {
        case kComponentOpenSelect:
        case kComponentCloseSelect:
        case kComponentCanDoSelect:
        case kComponentVersionSelect:
        case kComponentTargetSelect:
        case kMediaInitializeSelect:
        case kMediaIdleSelect:
        case kMediaSetDimensionsSelect:
        case kMediaSetMatrixSelect:
            return true;
        default:
            return ComponentFunctionImplemented
                            (globals->delegateComponent, ftnNumber);
    }
}

pascal ComponentResult PictVersion (PictGlobals globals)
{
    return 0x00020001;
}

pascal ComponentResult PictOpen(PictGlobals globals,
                                        ComponentInstance self)
{
    OSErr err;

    /* allocate storage */
    globals = (PictGlobals)NewPtrClear(sizeof(PictGlobalsRecord));
```

```
    if (err = MemError()) return err;
    SetComponentInstanceStorage(self, (Handle)globals);
    globals->self = self;
    globals->parent = self;

    /* find a base media handler to serve as a delegate */
    globals->delegateComponent =
            OpenDefaultComponent (MediaHandlerType,
                                        BaseMediaType);
    if (globals->delegateComponent)
        PictTarget(globals, self); /* set up the calling chain */
    else {
        DisposePtr((Ptr)globals);
        err = cantOpenHandler;
    }
    return err;
}

pascal ComponentResult PictClose (PictGlobals globals,
                                        ComponentInstance self)
{
    if (globals) {
        if (globals->delegateComponent)
            CloseComponent(globals->delegateComponent);
        DisposePtr((Ptr)globals);
    }
    return noErr;
}

pascal ComponentResult PictTarget(PictGlobals store,
                                      ComponentInstance parentComponent)
{
    /* remember who is at the top of your calling chain */
    store->parent = parentComponent;

    /* and inform your delegate component of the change */
    ComponentSetTarget(store->delegateComponent, parentComponent);

    return noErr;
}
```

## Initializing a Derived Media Handler Component

The derived media handler component is initialized by the Movie Toolbox's calling of the `MediaInitialize` function. You should then report the derived media handler capabilities to the base media handler before the Movie Toolbox starts working with your media by calling the `MediaSetHandlerCapabilities` function from your `MediaInitialize` function.

Listing 2-2 is the initialization function for a derived media handler. The `PictInitialize` function stores the initial height, width, track movie matrix, media, and track of the derived media handler component. From `PictInitialize`, the `MediaSetHandlerCapabilities` function is called to inform the base media handler of its existence and features.

**Listing 2-2**     Initializing a derived media handler

```
pascal ComponentResult PictInitialize (PictGlobals store,
                                       GetMovieCompleteParams *gmc)
{
    /* remember some useful parameters */
    store->width = gmc->width;
    store->height = gmc->height;
    store->matrix = gmc->trackMovieMatrix;
    store->media = gmc->theMedia;
    store->track = gmc->theTrack;

    /* tell the base media handler about your derived
        media handler */
    MediaSetHandlerCapabilities(store->delegateComponent,
    handlerHasSpatial, handlerHasSpatial);

    return noErr;
}
```

## Drawing the Media Sample

The Movie Toolbox provides processing time to your derived media handler to display samples by calling the `MediaIdle` function. Your media handler may use this time to play its media sample. The code in Listing 2-3 allows the derived media handler component to draw the current media sample (in this case, a PICT image).

**Listing 2-3**     Drawing the media sample

```
pascal ComponentResult PictIdle (PictGlobals store,
                                 TimeValue atMediaTime,
                                 long flagsIn, long *flagsOut,
                                 const TimeValue *tr)
{
    OSErr err;
    Rect r;
    Handle sample = NewHandle (0);
    if (err = MemError()) goto bail;

    /* get the current sample */
    err = GetMediaSample (store->media, sample, 0, nil,
                          atMediaTime, nil, 0, 0, 0, 0, 0, 0);
    if (err) goto bail;

    /* draw it using the current matrix */
    SetRect (&r, 0, 0, FixRound (store->width),
             FixRound (store->height));
    TransformRect (&store->matrix, &r, nil);
    EraseRect (&r);
    DrawPicture ((PicHandle)sample, &r);

bail:
    DisposeHandle (sample);
    *flagsOut |= mDidDraw;           /* let Movie Toolbox know you drew
                                        something */
    return err;
```

```
}

pascal ComponentResult PictSetDimensions (PictGlobals store,
                                          Fixed width,
                                          Fixed height)
{
    /* remember the new track */
    store->width = width;
    store->height = height;
    return noErr;
}

pascal ComponentResult PictSetMatrix (PictGlobals store,
                                      MatrixRecord *trackMovieMatrix)
{
    /* remember the new display matrix */
    store->matrix = *trackMovieMatrix;
    return noErr;
}
```

# Request Processing

Because your derived media handler is based on the base media handler component, you avoid many of the details involved in creating a media handler. However, your derived media handler must observe a few rules when processing service requests. These rules are as follows:

■   When you receive an open request from the Component Manager, in addition to the other processing you perform on your own behalf, you must also open a connection to the base media handler component. You should save the component instance that is returned by the Component Manager so that your media handler can use the services of the base media handler.

■   The base media handler has a component type of `MediaHandlerType` (which is set to `'mhlr'`) and a component subtype of `BaseMediaType` (which is set to `'gnrc'`). You can use these values with the Component Manager's `OpenDefaultComponent` function to open a connection to the base media handler.

■   At this time, you must also tell the base media handler that your handler is derived from it. Use the Component Manager's `OpenComponent` function to create a component instance of your media handler as a descendant of the base media handler. After calling that function, you should send the `kComponentSetTargetSelect` request to the base media handler, so that it knows your media handler is derived from it. Use the Component Manager's `ComponentSetTarget` function to send a target request.

■   When you receive a close request from the Component Manager, be sure to close your handler's connection to the base media handler component. Use the Component Manager's `CloseComponent` function.

■   Your derived media handler must support the target request, so that your component can be used by other media handlers.

■   Be sure to pass all unsupported service requests to the base media handler component. Use the Component Manager's `DelegateComponentCall` function to pass these requests to the base media handler.

■   If your media handler component competes for potentially scarce system resources, your component should release those resources when you aren't using them. For example, if you are creating a media handler that uses sound, you might use sound channels. Because there are a limited number of sound channels available, your component should free its channels whenever your media is not playing or has been stopped. You can reallocate the channels when you start playing or your component's `MediaPreroll` function is called.

# Derived Media Handler Support

This chapter describes the functions and constants that your derived media handler may support and the data structure that your component may use to interact with the base media handler.

## Data Structure

The `GetMovieCompleteParams` data type defines the layout of the complete movie parameter structure used by the `MediaInitialize` function:

```
typedef struct {
    short               version;                /* version; always 1 */
    Movie               theMovie;               /* movie identifier */
    Track               theTrack;               /* track identifier */
    Media               theMedia;               /* media identifier */
    TimeScale           movieScale;             /* movie's time scale */
    TimeScale           mediaScale;             /* media's time scale */
    TimeValue           movieDuration;          /* movie's duration */
    TimeValue           trackDuration;          /* track's duration */
    TimeValue           mediaDuration;          /* media's duration */
    Fixed               effectiveRate;          /* media's effective rate */
    TimeBase            timeBase;               /* media's time base */
    short               volume;                 /* media's volume */
    Fixed               width;                  /* width of display area */
    Fixed               height;                 /* height of display area */
    MatrixRecord        trackMovieMatrix;       /* transformation matrix */
    CGrafPtr            moviePort;              /* movie's graphics port */
    GDHandle            movieGD;                /* movie's graphics device */
    PixMapHandle        trackMatte;             /* track's matte */
    QTAtomContainer     inputMap;               /* media's input map */
} GetMovieCompleteParams;
```

| Field | Description |
|---|---|
| version | Specifies the version of this structure. This field is always set to 1. |
| theMovie | Identifies the movie that contains the current media's track. This movie identifier is supplied by the Movie Toolbox. Your component may use this identifier to obtain information about the movie that is using your media. |
| theTrack | Identifies the track that contains the current media. This track identifier is supplied by the Movie Toolbox. Your component may use this identifier to obtain information about the track that contains your media. For example, you might call the Movie Toolbox's `GetTrackNextInterestingTime` function in order to examine the track's edit list. |

| Field | Description |
|---|---|
| `theMedia` | Identifies the current media. This media identifier is supplied by the Movie Toolbox. Your derived media handler can use this identifier to read samples or sample descriptions from the current media, using the Movie Toolbox's `GetMediaSample` and `GetMediaSampleDescription` functions. |
| `movieScale` | Specifies the time scale of the movie that contains the current media's track. If the Movie Toolbox changes the movie's time scale, the toolbox calls your derived media handler's `MediaSetMovieTimeScale` function. |
| `mediaScale` | Specifies the time scale of the current media. If the Movie Toolbox changes your media's time scale, the toolbox calls your derived media handler's `MediaSetMediaTimeScale` function. |
| `movieDuration` | Contains the movie's duration. This value is expressed in the movie's time scale. |
| `trackDuration` | Contains the track's duration. This value is expressed in the movie's time scale. |
| `mediaDuration` | Contains the media's duration. This value is expressed in the media's time scale. |
| `effectiveRate` | Contains the media's effective rate. This rate ties the media's time scale to the passage of absolute time, and does not necessarily correspond to the movie's rate. This value takes into account any master time bases that may be serving the media's time base. The value of this field indicates the number of time units (in the media's time scale) that pass each second. This rate is represented as a 32-bit, fixed-point number. The high-order 16 bits contain the integer portion, and the low-order 16 bits contain the fractional portion. The rate is negative when time is moving backward for the media. Whenever the Movie Toolbox changes your media's effective rate, it calls your derived media handler's `MediaSetRate` function. |
| `timeBase` | Identifies the media's time base. |
| `volume` | Contains the media's current volume setting. This value is represented as a 16-bit, fixed-point number. The high-order 8 bits contain the integer portion; the low-order 8 bits contain the fractional part. Volume values range from -1.0 to 1.0. Negative values play no sound but preserve the absolute value of the volume setting. If the Movie Toolbox changes your media's volume, it calls your derived media handler's `MediaGSetVolume` function. |
| `width` | Indicates the width, in pixels, of the track rectangle. This field, along with the `height` field, specifies a rectangle that surrounds the image that is displayed when the current media is played. This value corresponds to the x coordinate of the lower-right corner of the rectangle and is expressed as a fixed-point number. If the Movie Toolbox modifies this rectangle, the toolbox calls your derived media handler's `MediaSetDimensions` function. Note that your media need not present only a rectangular image. The Movie Toolbox can use a clipping region to cause your media's image to be displayed in a region of arbitrary shape, and it can use a matte to control the image's transparency. The toolbox calls your derived media handler's `MediaSetClip` function whenever it changes your media's clipping region. The `trackMatte` field in this structure specifies a matte region. |

| Field | Description |
|---|---|
| `height` | Indicates the height, in pixels, of the track rectangle. This value corresponds to the y coordinate of the lower-right corner of the rectangle and is expressed as a fixed-point number. |
| `trackMovieMatrix` | Specifies the matrix that transforms your media's pixels into the movie's coordinate system. The Movie Toolbox obtains this matrix by concatenating the track matrix and the movie matrix. You should use this matrix whenever you are displaying graphical data from your media. Whenever the Movie Toolbox modifies this matrix, it calls your derived media handler's `MediaSetMatrix` function. |
| `moviePort` | Indicates the movie's graphics port. Whenever the Movie Toolbox changes the movie's graphics world, it calls your derived media handler's `MediaSetGWorld` function. |
| `movieGD` | Specifies the movie's graphics device. Whenever the Movie Toolbox changes the movie's graphics world, it calls your derived media handler's `MediaSetGWorld` function. |
| `trackMatte` | Identifies the matte region assigned to the track that uses your media. This field contains a handle to a pixel map that contains a blend matte. Your component is not responsible for disposing of this matte. If there is no matte, this field is set to `nil`. |
| `inputMap` | A reference to the media's input map. The media input map should not be modified or disposed. |

# Component Manager Flag

The Component Manager allows you to specify information about your component's capabilities in the `componentFlags` field of the component description record. Within this field, the `mediaHandlerFlagBaseClient` flag indicates that your component is derived from another component. Setting this flag to 1 tells the Component Manager that your component is a client of the base media handler.

# Functions

This section lists the functions that may be supported by derived media handler components.

> **Note:** Many of the functions described in this section are optional; your derived media handler may not need to support them. The description of each function discusses the issues you should consider when deciding whether or not to support a specific function.

## Managing the Component

Derived media handlers provide three functions that allow the Movie Toolbox to manage its relationship with the media handler. The Movie Toolbox calls your `MediaInitialize` to give you an opportunity to prepare to provide access to your media. The Movie Toolbox grants processing time to your handler by calling your `MediaIdle` function. Your `MediaGGetStatus` function allows the Movie Toolbox to retrieve status information after calling `MediaIdle`.

## General Data Management

While the base media handler isolates your component from the details of media data access, your derived media handler still needs to keep track of certain information in order to support movie playback and creation. This section discusses functions that help your media handler manage its information.

Your media handler may store proprietary information in its media. The Movie Toolbox calls two media handler functions in order to give you an opportunity to retrieve or store this information. The `MediaPutMediaInfo` function allows you to store your special information in your media. The `MediaGetMediaInfo` function delivers that data to your media handler.

The Movie Toolbox tells your media handler when its track has been enabled or disabled by calling your `MediaSetActive` function. The Movie Toolbox prepares your handler for playback by calling your `MediaPreroll` function. Whenever your media's playback rate changes, the Movie Toolbox calls your `MediaSetRate` function. Whenever the track that uses your media is edited, the Movie Toolbox calls your `MediaTrackEdited` function.

If the Movie Toolbox has called its `SetMediaSampleDescription` function on a sample description, it uses the `MediaSampleDescriptionChanged` function to notify your media handler of the change.

The Movie Toolbox allows tracks to be identified by various characteristics. For instance, it is possible to request that all tracks containing audio information be searched. To determine whether a track has a given characteristic, the Movie Toolbox queries the media handler for each track. The Movie Toolbox calls the `MediaHasCharacteristic` function with the specified characteristic.

The Movie Toolbox uses two functions to inform you about changes to your media's time environment. The `MediaSetMediaTimeScale` function allows the Movie Toolbox to change your media's time scale. The `MediaSetMovieTimeScale` function allows the Movie Toolbox to tell you when the movie's time scale has changed.

Other useful functions include

- `MediaGSetActiveSegment`
- `MediaInvalidateRegion`
- `MediaGetNextStepTime`
- `MediaTrackReferencesChanged`

- `MediaTrackPropertyAtomChanged`

- `MediaSetTrackInputMapReference`

- `MediaGetSampleDataPointer`

- `MediaReleaseSampleDataPointer`

- `MediaCompare`

- `MediaSetVideoParam`

- `MediaGetVideoParam`

- `MediaSetNonPrimarySourceData`

- `MediaGetOffscreenBufferSize`

- `MediaSetHints`

- `MediaGetName`

## Graphics Data Management

If your media handler draws media data on the screen, you need to manage your media's graphics environment. The Movie Toolbox uses a number of functions to inform you about changes to the graphics environment. The Movie Toolbox only calls these functions if you have set the `handlerHasSpatial` flag to 1 in the flags parameter of the `MediaSetHandlerCapabilities` function.

The Movie Toolbox calls your handler's `MediaSetGWorld` function whenever your media's graphics port or graphics device has changed. The `MediaSetDimensions` function allows the Movie Toolbox to inform your handler about changes to its spatial dimensions. Whenever either the movie or track matrix changes, the Movie Toolbox calls your `MediaSetMatrix` function. Similarly, if your media's clipping region changes, the Movie Toolbox calls your `MediaSetClip` function.

When it is building up a movie's image from its component tracks, the Movie Toolbox must be able to determine which tracks are transparent. The Movie Toolbox calls your `MediaGetTrackOpaque` function to retrieve this information about your media.

The Movie Toolbox calls your `MediaGetNextBoundsChange` function so that it can learn when your media will next change its display shape. When the Movie Toolbox wants to find out the shape of the region into which you draw your media, it calls your `MediaGetSrcRgn` function.

Other useful functions include

- `MediaGetDrawingRgn`

- `MediaGetGraphicsMode`

- `MediaSetGraphicsMode`

## Sound Data Management

The Movie Toolbox uses your `MediaGSetVolume` function to tell your media handler when its sound volume has changed. It uses `MediaSetSoundLocalizationData` to support 3D sound capabilities in a media handler that plays sound.

## Base Media Handler Utility Function

Apple's base media handler component provides a utility function, `MediaSetHandlerCapabilities`, which allows you to tell the base handler what your derived handler can do.

## Management of Progressive Downloads

The function `MediaMakeMediaTimeTable` is called by the base media handler to create a media time table whenever an application or other software calls the Toolbox's `QTMovieNeedsTimeTable`, `GetMaxLoadedTimeInMovie`, `MakeTrackTimeTable`, or `MakeMediaTimeTable` function. When an application or other software calls one of these functions, it allocates an unlocked relocatable memory block for the time table to be returned and passes a handle to it in the offsets parameter. Your derived media handler must resize the block to accommodate the time table it returns.

The time table your derived media handler returns is a two-dimensional array of long integers that is organized so that each row in the table contains values for one data reference. The first column in the table contains values for the time in the media specified by the `startTime` parameter, and each subsequent column contains values for the point in the media that is later by the value specified by the `timeIncrement` parameter. Each long integer value in the table specifies the offset, in bytes, from the beginning of the data reference for that point in the media.

# Constants

The constants listed in this section support the functions listed in Functions (page 21).

## Function Flags

```
/* flags in flags parameter of MediaSetHandlerCapabilities function */
enum {
    handlerHasSpatial            =   1<<0,   /* draws */
    handlerCanClip               =   1<<1,   /* clips */
    handlerCanMatte              =   1<<2,   /* reserved */
    handlerCanTransferMode       =   1<<3,   /* does transfer modes */
    handlerNeedsBuffer           =   1<<4,   /* use offscreen buffer */
    handlerNoIdle                =   1<<5,   /* never draws */
    handlerNoScheduler           =   1<<6,   /* schedules self */
    handlerWantsTime             =   1<<7,   /* needs more time */
    handlerCGrafPortOnly         =   1<<8    /* color only */
};

/* values for inFlags parameter of MediaIdle function */
enum {
    mMustDraw                    =      1<<3,   /* must draw now */
    mAtEnd                       =      1<<4,   /* current time
                                      corresponds to end of movie */
    mPreflightDraw               =      1<<5    /* must not draw */
};

/* values for outFlags parameter of MediaIdle function */
```

```
enum {
    mDidDraw                        =       1<<0,   /* did draw */
    mNeedsToDraw                    =       1<<2    /* needs to draw */
};
```

## Component Types and Characteristics

```
/* component type and subtype values */
#define MediaHandlerType          'mhlr'    /* derived media handler */
#define BaseMediaType             'gnrc'    /* base media handler */

/* constants used in the characteristic parameter of the
    MediaHasCharacteristic function */
#define VisualMediaCharacteristic 'eyes'   /* visual media characteristic */
#define AudioMediaCharacteristic  'ears'   /* audio media characteristic */
```

## Selectors

```
/* selectors for derived media handler components */
enum {
    kMediaInitializeSelect              = 0x501,  /* MediaInitialize */
    /* MediaSetHandlerCapabilities */
    kMediaSetHandlerCapabilitiesSelect  = 0x502,
    kMediaIdleSelect                    = 0x503,  /* MediaIdle */
    kMediaGetMediaInfoSelect            = 0x504,  /* MediaGetMediaInfo */
    kMediaPutMediaInfoSelect            = 0x505,  /* MediaPutMediaInfo */
    kMediaSetActiveSelect               = 0x506,  /* MediaSetActive */
    kMediaSetRateSelect                 = 0x507,  /* MediaSetRate */
    kMediaGGetStatusSelect              = 0x508,  /* MediaGGetStatus */
    kMediaTrackEditedSelect             = 0x509,  /* MediaTrackEdited */
    kMediaSetMediaTimeScaleSelect       = 0x50A,  /* MediaSetMediaTimeScale */
    kMediaSetMovieTimeScaleSelect       = 0x50B,  /* MediaSetMovieTimeScale */
    kMediaSetGWorldSelect               = 0x50C,  /* MediaSetGWorld */
    kMediaSetDimensionsSelect           = 0x50D,  /* MediaSetDimensions */
    kMediaSetClipSelect                 = 0x50E,  /* MediaSetClip */
    kMediaSetMatrixSelect               = 0x50F,  /* MediaSetMatrix */
    kMediaGetTrackOpaqueSelect          = 0x510,  /* MediaGetTrackOpaque */
    kMediaSetGraphicsModeSelect         = 0x511,  /* MediaSetGraphicsMode */
    kMediaGetGraphicsModeSelect         = 0x512,  /* MediaGetGraphicsMode */
    kMediaGSetVolumeSelect              = 0x513,  /* MediaGSetVolume */
    kMediaSetSoundBalanceSelect         = 0x514,  /* MediaSetSoundBalance */
    kMediaGetSoundBalanceSelect         = 0x515,  /* MediaGetSoundBalance */
    kMediaGetNextBoundsChangeSelect     = 0x516, /* MediaGetNextBoundsChange */
    kMediaGetSrcRgnSelect               = 0x517,  /* MediaGetSrcRgn */
    kMediaPrerollSelect                 = 0x518,  /* MediaPreroll */
    /* MediaSampleDescriptionChanged */
    kMediaSampleDescriptionChangedSelect = 0x519,
    kMediaHasCharacteristicSelect       = 0x51A  /* MediaHasCharacteristic */
};
```

## Media Video Parameters

The `whichparam` parameter to the `MediaSetVideoParam` and `MediaGetVideoParam` functions specifies which video parameter you want to adjust. QuickTime defines these constants that you can use to configure the `whichparam` parameter.

```
enum {
    kMediaVideoParamBrightness  = 1,
    kMediaVideoParamContrast    = 2,
    kMediaVideoParamHue         = 3,
    kMediaVideoParamSharpness   = 4,
    kMediaVideoParamSaturation  = 5,
    kMediaVideoParamBlackLevel  = 6,
    kMediaVideoParamWhiteLevel  = 7
};
```

| Term | Definition |
|------|------------|
| `kMediaVideoParam-Brightness` | The brightness value controls the overall brightness of the digitized video image. Brightness values range from 0 to 65,535, where 0 is the darkest possible setting and 65,535 is the lightest possible setting. |
| `kMediaVideoParam-Contrast` | The contrast value ranges from 0 to 65,535, where 0 represents no change to the basic image and larger values increase the contrast of the video image (that is, increase the slope of the transform). |
| `kMediaVideoParamHue` | Hue is similar to the tint control on a television. It is specified in degrees with complementary colors set 180 degrees apart (red is 0 degrees, green is +120 degrees, and blue is -120 degrees). QuickTime supports hue values that range from 0 (-180 degrees shift in hue) to 65,535 (+179 degrees shift in hue), where 32,767 represents a 0 degrees shift in hue. |
| `kMediaVideoParam-Sharpness` | The sharpness value ranges from 0 to 65,535, where 0 represents no sharpness filtering and 65,535 represents full sharpness filtering. Higher values result in a visual impression of increased picture sharpness |
| `kMediaVideoParam-Saturation` | The saturation value controls color intensity. For example, at high saturation levels, red appears to be red; at low saturation, red appears pink. Valid saturation values range from 0 to 65,535, where 0 is the minimum saturation value and 65,535 specifies maximum saturation. |
| `kMediaVideoParam-BlackLevel` | Black level refers to the degree of blackness in an image.The highest setting produces an all-black image; on the other hand, the lowest setting yields little, if any, black even with black objects in the scene. Black level values range from 0 to 65,535, where 0 represents the maximum black value and 65,535 represents the minimum black value. |
| `kMediaVideoParam-WhiteLevel` | White level refers to the degree of whiteness in an image. White level values range from 0 to 65,535, where 0 represents the minimum white value and 65,535 represents the maximum white value |

# Preview Components

This chapter describes what preview components are and what they do.

Preview components provide two basic services: they draw and create previews. This section describes how preview components obtain preview data, what kind of information is stored with the file, and what they do with the preview data.

## Obtaining Preview Data

Preview components obtain data from

- a small data cache

- a reference they create to another resource in the file

- the file for which they are invoked

The preview component can create a small data cache containing the preview. Although creation of the preview cache may be time-consuming, the cache can then be stored in the file and used to display the preview for the file rapidly on subsequent occasions. The picture file preview component, which creates a thumbnail picture for the file and stores it in the file's resource fork, is one way of getting information from a data cache.

The preview component can create a reference to another resource in the file. For example, some file types already contain a picture preview in them. The preview component can then create a pointer to that existing data, rather than making another copy of it. The movie preview component works in this way when the preview for the movie is actually the movie's preview, rather than only its poster picture.

If the preview component can display the preview for the file quickly enough in every case, there is no need for a cache. Such a preview component reinterprets the data in the file each time it is invoked, rather than creating a preview cache once. This method of getting the information allows the file to remain untouched, requires no disk space, and does not demand that the user or the application make any special effort to create the preview. Unfortunately, in most cases, it is not possible to interpret the data quickly enough to use this approach. Preview components that handle this type of preview should set the `pnotComponentNeedsNoCache` flag in their component flags field.

```
enum {
    pnotComponentNeedsNoCache = 2
};
```

If a preview component relies on other system software services, it must make sure they are present. For example, if your preview component uses the Movie Toolbox, it is responsible for calling the Movie Toolbox's `EnterMovies` and `ExitMovies` functions.

When previewing is complete, the component receives a normal Component Manager close request. If you add any controls to the window, you should dispose of them while you are calling the Component Manager's `CloseComponent` function.

A preview component should never write back to the file directly. The caller of the preview component is responsible for actually modifying the file. You should open all access paths to the file with read permission only.

Figure 4-1 illustrates the relationships of a preview component, the Image Compression Manager, and an application.
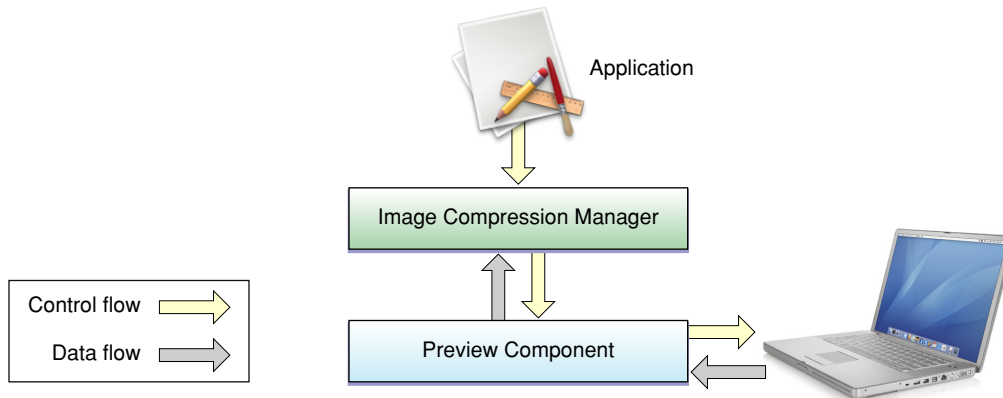


## Storing Preview Data in Files

A preview may or may not contain sound or text data or other types of information. In addition to the visual preview, QuickTime provides the preview resource, described on  The Preview Resource (page 28), which also allows you to store

■  a brief description of the file

■  a list of keywords

■  an associated language code to allow use of a single file in more than one region

■  a modification date to help applications determine when the data has been changed

## The Preview Resource

QuickTime uses the preview resource (defined by the `pnotResource` data type) with a resource ID of 0 to store the visual preview information. The structure of the preview resource is shown in Listing 4-1.

> ⚠️ **Warning:**  If you parse this resource directly, please do extensive error checking in your code so as not to hinder future expansion of the data structure. In particular, if you encounter unknown version bits, exercise caution. Unexpected results may occur.

**Listing 4-1**     The preview resource

```
typedef struct pnotResource {
    unsigned long    modDate;    /* modification date */
    short            version;    /* version number of preview resource */
    OSType           resType;    /* type of resource used as preview cache */
    short            resID;      /* resource identification number
                                    of resource used as preview cache */
    short            numResItems; /* number of additional file
                                    descriptions */
    pnotResItem      resItem[ ];  /* array of file descriptions */
} pnotResource;
```

| Field | Description |
|---|---|
| modDate | Contains the modification time (in standard Macintosh seconds since midnight, January 1, 1904) of the file for which the preview was created. This parameter allows you to find out if the preview is out of date with the contents of the file. |
| version | Contains the version number of the preview resource. The low bit of the version is a flag for preview components that only reference their data. If the bit is set, it indicates that the resource identified in the preview resource is not owned by the preview component, but is part of the file. It is not removed when the preview is updated or removed (using the Image Compression Manager's `MakeFilePreview` or `AddFilePreview` function), as it would be if the version number were 0. |
| resType | Contains the type of a resource used as a preview cache for the given file. The type of the resource determines the subtype of the preview component that should be used to display the preview. |
| resID | Contains the identification number of a resource used as a preview cache for the specified file. |
| numResItems | Specifies the number of additional file descriptions stored with this preview. |
| resItem | Contains the preview resource item structure (defined by the `pnotResItem` data type). |

# Using the Preview Data

Preview components may

- create a preview

- draw a preview

- create and draw a preview

Some preview components only create a preview and rely on another component to display it. For example, by default, the movie preview component creates a picture preview for the file. This is displayed by the picture preview component.

Most preview components simply draw the preview. These are the simplest type of display components. They do not require any other event processing (including the scheduling of idle time) to play a movie. The picture preview component is an example of this type of component.

Preview components that do not require a cache should have a subtype that matches the type of file for which they can display previews.

A preview component for sound would require event processing, since it would need time to play the sound. If your preview component requires event processing, you must have the `pnotComponentWantsEvents` flag set in its component flags field.

```
enum {
    pnotComponentWantsEvents = 1,
};
```

# Creating Preview Components

This chapter describes how to create your own preview component. A listing of a sample component is included.

## Overview

Preview components that create previews have a type of `'pmak'` and a subtype that matches the type of the file for which they create previews.

Preview components that display previews have a type of `'pnot'` and a subtype that matches the type of the resource that they display.

You can use the following constants to refer to the request codes for each of the functions that your preview component must support.

```
enum {
  kPreviewShowDataSelector            = 1, /* PreviewShowData */
  kPreviewMakePreviewSelector         = 2, /* PreviewMakePreview */
  kPreviewMakePreviewReferenceSelector = 3, /* PreviewMakePreviewReference */
  kPreviewEventSelector               = 4  /* PreviewEvent */
};
```

This section presents a sample program that displays a preview component for the display of PICS animation files. First it implements the required Component Manager functions. Then it converts the PICT image data into a format for display as a preview.

## Implementing Required Component Functions

Listing 5-1 supplies the component dispatchers for the preview component together with the can do, version, open, and close functions.

**Listing 5-1**     Implementing the required Component Manager functions

```
typedef struct {
    ComponentInstance       self;
} PICSPreviewRecord, **PICSPreviewGlobals;

/* entry point for all Component Manager requests */
pascal ComponentResult PICSPreviewDispatch
                        (ComponentParameters *params, Handle store)
{
    OSErr err = badComponentSelector;
    ComponentFunction componentProc = 0;
```

```
    switch (params->what) {
        case kComponentOpenSelect:
            componentProc = PICSPreviewOpen; break;
        case kComponentCloseSelect:
            componentProc = PICSPreviewClose; break;
        case kComponentCanDoSelect:
            componentProc = PICSPreviewCanDo; break;
        case kComponentVersionSelect:
            componentProc = PICSPreviewVersion; break;
        case kPreviewShowDataSelector:
            componentProc = PICSPreviewShowData; break;
    }

    if (componentProc)
        err = CallComponentFunctionWithStorage (store, params,
                                                componentProc);

    return err;
}

pascal ComponentResult PICSPreviewCanDo
                        (PICSPreviewGlobals store, short ftnNumber)
{
    switch (ftnNumber) {
        case kComponentOpenSelect:
        case kComponentCloseSelect:
        case kComponentCanDoSelect:
        case kComponentVersionSelect:
        case kPreviewShowDataSelector:
            return true;
        default:
            return false;
    }
}

pascal ComponentResult PICSPreviewVersion
                        (PICSPreviewGlobals store)
{
    return 0x00010001;
}

pascal ComponentResult PICSPreviewOpen (PICSPreviewGlobals store,
                                        ComponentInstance self)
{
    store = (PICSPreviewGlobals)NewHandle
                        (sizeof (PICSPreviewRecord));
    if (!store) return MemError();
    SetComponentInstanceStorage (self, (Handle)store);
    (**store).self = self;

    return noErr;
}

pascal ComponentResult PICSPreviewClose
                        (PICSPreviewGlobals store,
                         ComponentInstance self)
{
    if (store) DisposeHandle ((Handle)store);
```

```
        return noErr;
    }
```

# Displaying Image Data as a Preview

To display a file's image preview, your `PreviewShowData` function is called. Listing 5-2 includes the `PICSPreviewShowData` function, which previews a PICS file. The function loads the first PICT image from the PICS file and uses the PICT file preview component to display it.

**Listing 5-2**     Converting data into a form that can be displayed as a preview

```
pascal ComponentResult PICSPreviewShowData(
            PICSPreviewGlobals store, OSType dataType, Handle data,
            const Rect *inHere)
{
    OSErr err = noErr;
    short resRef = 0, saveRes = CurResFile();
    FSSpec theFile;
    Boolean whoCares;
    Handle thePict = nil;
    ComponentInstance ci;

    /* because your component has the pnotComponentNeedsNoCache
       flag set, it should only be called to display files */
    if (dataType != rAliasType)
        return paramErr;

    /* open up the file to preview */
    if (err = ResolveAlias (nil, (AliasHandle)data, &theFile,
                            &whoCares)) goto bail;
    resRef = FSpOpenResFile (&theFile, fsRdPerm);
    if (err = ResError()) goto bail;

    /* get the first 'PICT' */
    UseResFile (resRef);
    thePict = Get1IndResource ('PICT', 1);
    if (!thePict) goto bail;

    /* use the PICT preview component to display the preview */
    if (ci = OpenDefaultComponent (ShowFilePreviewComponentType, 'PICT'))
    {
        PreviewShowData (ci, 'PICT', thePict, inHere);
        CloseComponent (ci);
    }

bail:
    if (resRef) CloseResFile (resRef);
    if (thePict) DisposeHandle (thePict);
    UseResFile (saveRes);
    return err;
}
```

# Functions For Displaying Previews

## Introduction

This chapter describes the functions for displaying previews, handling events in previews, and creating previews that are provided by preview components. These functions are described from the perspective of the Image Compression Manager, which is most likely to call preview components. If you are developing a preview component, your component must behave as described here.

## Handling Events

The `PreviewEvent` function is provided so that your preview component can do standard event filtering.

## Creating Previews

Two functions are available for use in creating previews. The `PreviewMakePreview` function creates previews by allocating a handle to data to be added to the file. On the other hand, the `PreviewMakePreviewReference` function makes previews by returning the type and identification number of a resource within the file to be used as the preview for the file.

## The Preview Resource

This section describes the preview resource, which is used to store a visual preview, and the preview resource item structure, which is an array that allows you to store additional preview information.

The preview display code assumes that the data fork of the file is formatted using QuickTime atoms. See *QuickTime Movie Basics* for information on atom-based storage.

Adding a preview results in at least two atoms being added to the data file. The first atom has a `pnot` tag. Its basic structure is the same as the `pnotResource` structure.

```
struct PreviewResourceRecord {
    unsigned long       modDate;
    short               version;
    OSType              resType;
    short               resID;
};
```

| Term | Definition |
|------|------------|
| `modDate` | Contains the modification time (in the standard Macintosh format of seconds since midnight, January 1, 1904) of the file for which the preview was created. This parameter allows you to find out if the preview is out of date with the contents of the file. |
| `version` | Contains the version number of the preview resource. The low bit of the version is a flag for preview components that only reference their data. If the bit is set, it indicates that the resource identified in the preview resource is not owned by the preview component, but is part of the file. It is not removed when the preview is updated or removed (using the Image Compression Manager's `MakeFilePreview` or `AddFilePreview` function), as it would if the version number were 0. |
| `resType` | Identifies the type of the preview component used to display the preview data and the type of the atom containing the preview data. |
| `resID` | Contains the index (1-based) of the atom to be used. For example, a `resType` of PICT and a `resID` of 2 tells QuickTime to use the second PICT atom in the file for the preview data. |

# Data Types

This section defines the component instance used by preview components, and lists the data structures for preview resources and the preview resource item structure. See The Preview Resource (page 28) above, which includes the new `PreviewResourceRecord` data structure.

```
typedef ComponentInstance pnotComponent;
typedef struct pnotResource {
    unsigned long  modDate;      /* modification date */
    short          version;      /* version number of preview resource */
    OSType         resType;      /* type of resource used as preview cache */
    short          resID;        /* resource identification number
                                    of resource used as preview cache */
    short          numResItems;  /* number of additional file descriptions */
    pnotResItem    resItem[ ];   /* array of file descriptions */
} pnotResource;

typedef struct pnotResItem {
    unsigned long      modDate;    /* last modification date of item */
    OSType             useType;    /* what type of data */
    OSType             resType;    /* resource type containing item */
    short              resID;      /* resource ID containing this item */
    short              rgnCode;    /* region code */
    long               reserved;   /* set to 0 */
} pnotResItem; *pnotResItemPtr;
```

# Constants

This section defines the constants that are used to communicate with preview components. These are primarily flags that describe the preview components capabilities and requirements.

```
enum {
    pnotComponentWantsEvents    = 1, /* component requires events */
    pnotComponentNeedsNoCache   = 2  /* component does not require cache */
};
enum {
  kPreviewShowDataSelector            = 1, /* PreviewShowData */
  kPreviewMakePreviewSelector         = 2, /* PreviewMakePreview */
  kPreviewMakePreviewReferenceSelector = 3, /* PreviewMakePreviewReference */
  kPreviewEventSelector               = 4  /* PreviewEvent */
};
#define ShowFilePreviewComponentType 'pnot'    /* creates previews */
#define CreateFilePreviewComponentType 'pmak'  /* displays previews */
```

# Document Revision History

This table describes the changes to *QuickTime Component Creation Guide*.

| Date | Notes |
|---|---|
| 2007-01-08 | Revised artwork |
| 2006-01-10 | Replaces "Media Handlers: Creating Media Handler Components" and "Preview Components." |
| 2002-09-17 | New document that explains how to create a QuickTime media handler. |