QuickTime Transport and Delivery Guide

QuickTime > Transport & Delivery

2006-01-10

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Simultaneously published in the United States and Canada.

Contents

Listings

Introduction to QuickTime Transport and Delivery Guide

This book describes the principal ways that QuickTime transports and delivers data to specific devices, by using data handler components and by using video output components:

- Data handler components read and write movie data to specific devices, such as HFS disk files or computer memory. These are extremely low-level pieces of software, and are normally transparent to applications. Applications use data handler components indirectly, by making calls to the movie toolbox or a sequence grabber component. Applications can call some data handler component functions directly, however, for more complete control of data retrieval and storage. Apple provides data handler components for most device types. If you need to read or write data to a new or unsupported device type, you may need to create a data handler component.
- Video output components allow you to send QuickTime video to devices that are not recognized as displays by your computer's operating system. Video output components are used directly by applications that allow the user to send movie output to external devices. Manufacturers of video output hardware may need to create a video output component to use with their products. Applications use video output components by selecting a component, configuring it, and associating it with a graphics world.

Note: This book replaces two previously separate Apple documents: "Data Handler Components" and "Video Output Components."

You need to read this book if you plan to create a data handler component, a video output component, or a sequence grabber component. Sequence grabber components need to be able to select and use data handler components.

Applications programmers who think they may need to call data handler components directly should read the section Using Data Handler Components (page 11). The section About Data Handler Components (page 9) may also be of general interest to QuickTime developers.

Most developers should read the section Video Output Components (page 17) to understand what video output components are, and when they should be used. Applications developers who will use video output components should also read the section Using Video Output Components (page 23) and refer to the Functions Used To Control Video Output Components (page 27) section as necessary.

Organization of This Document

This book is divided into the following chapters:

- About Data Handler Components (page 9) describes what data handler components are, what they do, and how they work. Diagrams are included which illustrate their different use during movie playback and movie capture.

- Using Data Handler Components (page 11) describes how to use a data handler component. Developers writing sequence grabber components will use the interfaces described in this section. Developers writing data handler components will need to support these interfaces.
- Video Output Components (page 17) describes what video output components are, and what they do.
- Using Video Output Components (page 23) explains how to use video output components in your software.
- Functions Used To Control Video Output Components (page 27) discusses the functions used to control video output components.
- Creating Video Output Components (page 33) describes the routines you must implement when creating a video output component.
- Creating Data Handler Components (page 39) describes the requirements for creating a data handler component.

See Also

The following 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 Media Types and Media Handlers Guide* introduces the idea of QuickTime media handler components and provides details of the video, sound, text, timecode, and tween media handlers.
- *QuickTime Guide for Windows* provides information specific to programming for QuickTime on the Windows platform.
- *QuickTime API Reference* provides encyclopedic details of all the functions, callbacks, data types and structures, atom types, and constants in the QuickTime API.

About Data Handler Components

A data handler component stores and retrieves time-based data on a storage device, such as a movie file, on behalf of another QuickTime component, typically a media handler component or a sequence grabber component. Different QuickTime components are used depending on whether you are retrieving or storing data.

Movie Playback

During data retrieval, such as playback of a movie, a media handler component isolates your application and the Movie Toolbox from the details of how to retrieve data from a particular storage medium. Therefore, unless you are writing your own media handler, you do not have to directly use data handler components in your application; the retrieval of your data will be taken care of for you by the media handler the Movie Toolbox calls. However, you can call the data handler directly if you need to explicitly tell the data handler something, such as to use less memory when caching QuickTime data. If you are reading from a non-Macintosh storage medium, or multiple storage media, you might need to write your own data handler.

Movie Capture

During data storage, such as the capture of video and sound into a movie file, a a sequence grabber component isolates your application from the details of how to capture the raw data from a particular device. Therefore, during movie capture you do not have to directly use data handler components in your application. The storage of your data will be taken care of for you by the sequence grabber component you call. If, however, you are storing data onto a non-Macintosh or proprietary storage medium, or multiple storage media, you might need to write your own data handler.

The sequence grabber component calls the appropriate channel component, such as a video, sound, or text channel component, to retrieve the raw data from an input device, such as a microphone.

Processing data

Data handlers do not know anything about the content of the data they process. It is the responsibility of the client (that is, a media handler component or a channel component) to process the data. In the case of a movie's video data during movie playback, for example, the media handler takes the data from a data handler and uses the facilities of the Image Compression Manager to display the movie data on the computer screen.

While data handlers do not work with the content of the data they process, they must be aware of all of the details involved in storing and retrieving data from the storage medium they support. Apple provides several data handlers and a selection mechanism for choosing an appropriate handler. For example, one supports

data access from HFS volumes and another supports the memory-based data handler, which allows QuickTime to retrieve movies from memory handles. These two data handler components use very different mechanisms to store and retrieve movie data.

You might need to write your own data handler when you are accessing a storage medium for which there is no Apple-supplied data handler or when playing movies from a multimedia server, as you will need to use a data handler that understands the network protocols and data formats necessary to communicate with that server.

Identifying Containers With Data References

A **container** is the system element that contains the movie data and can be any element that can contain data. For example, a container may be an in-memory data structure, a local disk file, or a file on a networked multimedia server. As is the case throughout QuickTime, all data handlers identify their movie-data containers with data references. Data references identify the location of the container and its type.

Different container types may require different types of references. For example, files are identified using aliases, while memory-based movies are identified by handles. The data reference data type is flexible enough to accommodate all these cases. The data handler component must specify the type of reference it requires and verify that the references supplied by client applications are valid. Data handler components use the component subtype value to specify the reference type they support.

Whenever an application opens a container, the Movie Toolbox determines the most appropriate data handler component to use in order to access that container. The Movie Toolbox makes this determination by querying the various data handlers installed on the user's computer. If your application uses the Movie Toolbox, this selection process is transparent to your program. If you develop your own data handler, your component must support the selection functions.

Using Data Handler Components

This section describes how applications use data handler components. You should read this section if you are writing your own media handler or your own data handler.

Selecting a Data Handler

To help developers choose the best data handler for a specific situation while still making it easy for an application to find a usable data handler, Apple has defined two separate and complementary mechanisms for selecting data handler components. You can use the Component Manager's selection mechanisms to find a data handler that meets your needs, or you can interrogate a data handler to determine if it supports a specific data reference. Both mechanisms rely on characteristics of the current data reference in order to make the selection.

Before you can use a data handler component, your application must open a connection to that component. The easiest way to open a connection to a data handler component is to call the Movie Toolbox's `GetDataHandler` function. You supply a data reference and the Movie Toolbox selects an appropriate data handler component for you. This function is preferred for opening a data handler as it reliably chooses the best data handler.

Alternatively, you may use the Component Manager to open your connection. Call the Component Manager's `OpenDefaultComponent` or `OpenComponent` function to do so, but be aware that these functions are often unable to make the best choice when there are several different data handlers available for a file.

Selecting by Component Type Value

At the most basic level, your application can use the Component Manager's built-in selection mechanisms to find a data handler component for a data reference. You may use the Component Manager's `FindNextComponent` function in order to retrieve a list of all data handler components that meet your needs. You specify your request by supplying the component's characteristics in a component description record: in particular, in the `componentType`, `componentSubtype`, `componentManufacturer`, and `componentFlags` fields.

All data handler components have a component type value of `'dhlr'`, which is defined by the `dataHandlerType` constant. Data handler components use the value of the component subtype field to indicate the type of data reference they support. As a result of this convention, note that all data handlers that share a component subtype value must be able to recognize and work with data references of the same type. For example, file system data handlers always carry a component subtype value of `'alis'`, which indicates that their data references are file system aliases (note that this is true for QuickTime on Macintosh and under Windows, even though there is not, properly, a file system alias under Windows). Apple's memory-based data handler for Macintosh has a component subtype value of `'hndl'`.

Apple has not defined any special manufacturer field values or component flags values for data handler components. You may use the manufacturer field to select data handlers supplied by a specific vendor. To do so, you need to know the correct manufacturer field value for that vendor.

Interrogating a Data Handler's Capabilities

While you can use the Component Manager's selection mechanisms to find a data handler component that can recognize data references of a specific type, your application must interact with the data handler in order to determine whether it can support a specific data reference. Apple has defined two functions, `DataHCanUseDataRef` and `DataHGetVolumeList`, that allow you to query a data handler component in order to find out whether it can work with a data reference. By using these two functions, your application can choose a data handler that is best-suited to its specific needs.

Using the `DataHCanUseDataRef` function, you supply a data reference to the data handler component. The component then reports what it can do with that data reference. The returned value indicates the level and, to some extent, the quality of service the data handler can provide (for example, whether the component can read data from or write data to the data reference and whether the component uses any special support when working with that data reference).

Because calling the `DataHCanUseDataRef` function in several data handlers can be time consuming, Apple has also defined a function that helps narrow the search. By using the `DataHGetVolumeList` function, your application can obtain a list of all the file system volumes that a data handler can support. In response to your request, the data handler returns the list and flags indicating the level and quality of service the data handler can provide for containers on that volume.

For more information on these functions, see Determining Data Handler Capabilities (page 16).

Managing Data References

Once you have selected a data handler component, you must provide a data reference to the data handler. Use the `DataHSetDataRef` function to supply a data reference to a data handler. Once you have assigned a data reference to the data handler, your application may start reading or writing movie data from that data reference. The `DataHGetDataRef` function allows your application to obtain a data handler's current data reference.

Data handlers also provide a function that allows your application to determine whether two data references are equivalent (that is, refer to the same movie container). Your application provides a data reference to the `DataHCompareDataRef` function. The data handler returns a Boolean value indicating whether that data reference matches the data handler's current data reference.

For more information on these functions, see Working With Data References (page 13).

Working With Data References

All data handler components use data references to identify and locate a movie's container. Different types of containers may require different types of data references. For example, a reference to a memory-based movie may be a handle, while a reference to a file-based movie may be an alias.

Client programs can correlate data references with data handlers by matching the component's subtype value with the data reference type; the subtype value indicates the type of data reference the component supports. All data handlers with the same subtype value must support the same data reference type. To continue the previous example, Apple's memory-based data handler for Macintosh uses handles (and has a subtype value of `'hndl'`), while the HFS data handler uses Alias Manager aliases (its subtype value is `'alis'`).

The `DataHSetDataRef` and `DataHGetDataRef` functions allow applications to assign your data handler's current data reference. The `DataHCompareDataRef` function asks your component to compare a data reference against the current data reference and indicate whether the references are equivalent (that is, refer to the same container). The `DataHResolveDataRef` permits your component to locate a data reference's container.

The DataHSetOSFileRef and DataHGetOSFileRef functions provide an alternative, system-specific mechanism for assigning your data handler's current data reference.

For more information on data references, see Managing Data References (page 12).

Retrieving Movie Data

Before your application can read data using a data handler component, you must open a read path to the current data reference. Use the `DataHOpenForRead` function to request read access to the current data reference. Once you have gained read access to the data reference, data handlers provide both high- and low-level read functions.

The high-level function, `DataHGetData`, provides an easy-to-use, synchronous read interface. Being a synchronous function, `DataHGetData` does not return control to your application until the data handler has read and delivered the data you request.

If you need more control over the read operation, you can use the low-level function, `DataHScheduleData`, to issue asynchronous read requests. When you call this function, you provide detailed information specifying when you need the data from the request. The data handler returns control to your application immediately, and then processes the request when appropriate. When the data handler completes the request, it calls your data-handler completion function to report that the request has been satisfied, see Completion Function (page 14) for more information on the data-handler completion function.

Besides simply scheduling read requests that must be satisfied during a movie's playback, another use of the `DataHScheduleData` function is to prepare a movie for playback (commonly referred to as prerolling the movie). The `DataHScheduleData` function uses several special values to indicate a preroll operation. Your application calls the `DataHScheduleData` function one or more times to schedule the preroll read requests, and then uses the `DataHFinishData` function to tell the data handler to start delivering the requested data.

For more information on these functions and about preroll operations, see Reading Movie Data (page 14).

Completion Function

When client programs schedule asynchronous read or write operations (by calling your component's `DataHScheduleData` or `DataHWrite` functions), they furnish your component a data-handler completion function. Your component must call this function when it completes the read or write operation, whether the operation was a success or a failure.

Reading Movie Data

Data handler components provide two basic read facilities. The `DataHGetData` function is a fully synchronous read operation, while the `DataHScheduleData` function is asynchronous. Applications provide scheduling information when they call your component's `DataHScheduleData` function. When your component processes the queued request, it calls the application's data-handler completion function. By calling your component's `DataHFinishData` function, applications can force your component to process queued read requests. Applications may call your component's `DataHGetScheduleAheadTime` function in order to determine how far in advance your component prefers to get read requests.

Before any application can read data from a data reference, it must open read access to that reference by calling your component's `DataHOpenForRead` function. The `DataHCloseForRead` function closes that read access path.

For more information on reading movie data, see Retrieving Movie Data (page 13).

Storing Movie Data

Before your application can write data using a data handler component, you must open a write path to the current data reference. Use the `DataHOpenForWrite` function to request write access to the current data reference. Once you have gained write access to the data reference, data handler components provide both high- and low-level write functions.

Note: QuickTime for Windows version 2.1.1 or earlier does not support writing movie data.

The high-level function, `DataHPutData`, allows you to easily append data to the end of the container identified by a data reference. Except when capturing movie data using the sequence grabber component, the Movie Toolbox uses this call when writing data to movie files. However, this function does not allow your application to write to any location other than the end of the container. In addition, this is a synchronous operation, so control is not returned to your program until the write is complete. As a result, this function is not well-suited to high-performance write operations, such as would be required to capture a movie.

If you need a more flexible write facility, or one with higher performance characteristics, you can use the `DataHWrite` function. This function is intended to support high-speed writes, suitable for movie capture operations. For example, Apple's sequence grabber component uses this data handler function to capture movies.

When you call this function, you provide detailed information specifying the location in the container that is to receive the data. The data handler returns control to your application immediately, and then processes the request asynchronously. When the data handler completes the request, it calls your data-handler completion function to report that the request has been satisfied, see Completion Function (page 14) for more information on the data-handler completion function.

In addition to the `DataHWrite` function, data handler components provide several other "helper" functions that allow you to create new movie containers and prepare them for a movie capture operation.

For more information on all of these functions, see Writing Movie Data (page 15).

Writing Movie Data

As with reading movie data, data handlers provide two distinct write facilities. The `DataHPutData` function is a simple synchronous interface that allows applications to append data to the end of a container.

The `DataHWrite` function is a more capable, asynchronous write function that is suitable for movie capture operations. As is the case with the `DataHScheduleData` function, your component calls the application's data-handler completion function when you are done with the write request.

There are several other helper functions that allow applications to prepare your data handler for a movie capture operation. The `DataHCreateFile` function asks your component to create a new container. The `DataHSetFileSize` and `DataHGetFileSize` functions work with a container's size, in bytes. The `DataHGetFreeSpace` function allows applications to determine when to make a container larger. The `DataHPreextend` function asks your component to make a container larger. Applications may call your component's `DataHGetPreferredBlockSize` function in order to determine how best to interact with your data handler.

Before writing data to a data reference, applications must call your component's `DataHOpenForWrite` function to open a write path to the container. The `DataHCloseForWrite` function closes that write path.

Note that some data handlers may not support write operations. For example, some shared devices, such as a CD-ROM "jukebox," may be read-only devices. As a result, it is very important that your data handler correctly report its write capabilities to client programs. See Determining Data Handler Capabilities (page 16) for information about the functions that client programs use to interrogate your data handler. For more information on writing movie data, see Storing Movie Data (page 14).

Managing the Data Handler

Data handler components provide a number of functions that your application can use to manage its connection to the handler. The most important among these is `DataHTask`, which provides processor time to the handler. Your application should call this function often so that the handler has enough time to do its work.

Other functions in this category provide playback hints to the data handler and allow your application to influence how the component handles its cached data.

For more information on these functions, see Managing Data Handler Behavior (page 16).

Managing Data Handler Behavior

Applications may call your handler's `DataHPlaybackHints` function in order to provide you with some guidelines about how those applications play to use the current data reference.

The `DataHFlushData` and `DataHFlushCache` functions allow applications to influence how your component manages its stored data.

Determining Data Handler Capabilities

In order for client programs to choose the best data handler component for a data reference, Apple has defined some functions that allow applications to interrogate a data handler's capabilities.

The `DataHGetVolumeList` function allows an application to obtain a list of the volumes your data handler can support. The `DataHCanUseDataRef` function allows your data handler to examine a specific data reference and indicate its ability to work with the associated container. The `DataHGetDeviceIndex` function allows applications to determine whether different data references identify containers that reside on the same device.

By way of illustration, the Movie Toolbox uses the `DataHGetVolumeList` and `DataHCanUseDataRef` functions as follows. During startup, and whenever a new volume is mounted, the Movie Toolbox calls each data handler's `DataHGetVolumeList` function in order to obtain information about each handler's general capabilities. Specifically, the Movie Toolbox calls each component's `GetDataHandler`, `DataHGetVolumeList`, and `CloseComponent` functions.

Whenever an application opens a movie, the Movie Toolbox selects the best data handler for the movie's container. This may involve calling each appropriate data handler's `DataHCanUseDataRef` function (in some cases, a data handler may indicate that it does not need to examine a data reference before accessing it; see the description of the `DataHGetVolumeList` function for more information). For each data handler that can support the data reference (that is, has the correct component subtype value) and needs to be interrogated, the Movie Toolbox calls the component's `GetDataHandler`, `DataHCanUseDataRef`, and `CloseComponent` functions. Based on the resulting information, the Movie Toolbox selects the best data handler for the application.

Video Output Components

This section describes what video output components are, and what they do.

QuickTime video output, which most often comes from QuickTime movies, can be displayed in windows that appear on a computer's desktop. Because these windows are created and managed by the computer's operating system, software that presents QuickTime video can use the operating system's video display services to specify which display (when there is more than one video display) and window to use for video output.

There are, however, many video output devices that are not recognized by operating systems. To display QuickTime video on these devices, your software can use video output components. The components, which are normally developed by the manufacturers of video output devices, provide a standard interface for video output to a device.

How Video Output Components Process Video Data

A video output component receives QuickTime video data and delivers data to a video output device for display. If the incoming data is in a format that the video output device can display directly, the video output component can simply send the data to the video output device. If the incoming data cannot be displayed directly, the video output component must use a transfer codec or decompressor component to convert the data to a format that the video output device can display.

If a video output device cannot directly display 32-bit RGB data or data in one of the other supported QuickTime pixel formats, the developers of the device are strongly encouraged to provide a transfer codec that accepts data in one of the supported QuickTime pixel formats (preferably 32-bit RGB) and converts it to data that can be displayed on the device. When this transfer codec is available, any QuickTime video can be displayed on the video output device: the Image Compression Manager can convert any QuickTime images to a supported QuickTime pixel format and then invoke the transfer codec to display the result.

If any special decompressors, such as a transfer codec, are needed for a video output device, the decompressors are included in the definitions of the component's display modes, as described in Display Modes (page 17). How hardware developers can develop a transfer codec for their device is described in Creating a Transfer Codec for a Video Output Component (page 19).

Some video output devices do not accept pixels as input. For example, there are devices that display JPEG data directly. For these devices, a video output component can send the appropriate data directly, or it can invoke a compressor component to convert data in a pixel format to the appropriate data.

Display Modes

A video output device has one or more display modes. The characteristics of each mode determine how video is displayed. When any software displays video on a video output device, it must select which of the device's display modes to use.

The characteristics of a display mode include

- the height of the displayed image, in pixels
- the width of the displayed image, in pixels
- the horizontal resolution of the display, in pixels per inch
- the vertical resolution of the display, in pixels per inch
- the refresh rate of the display, in Hertz
- the pixel type of the display
- a text description of the display mode

The characteristics can also include a list of decompressor components required for the mode that are provided specifically for the video output device. If a video output device cannot directly display any of the pixel formats supported by QuickTime, the vendor of the device must provide one or more special decompressors to convert supported pixel formats to a format the device can display. If a video output device can display one or more of the pixel formats supported by QuickTime, the Image Compression Manager can use standard decompressors that are included with QuickTime, and the list of special decompressor components can be empty.

These characteristics, returned by the `QTVideoOutputGetDisplayModeList` function, are stored in a QT atom container. For a description of this QT atom container, see Display Mode QT Atom Container (page 29).

Transfer Codecs

If you are the manufacturer of a video output device, you need to provide a video output component for your device as described in Creating Video Output Components (page 33). In addition, if your video output device cannot display a pixel format defined by QuickTime, you should include a special decompressor, known as a transfer codec, that converts one of the supported QuickTime pixel formats (preferably 32-bit RGB) to data that the device can display. When this transfer codec is available, the QuickTime Image Compression Manager automatically uses it together with its built-in decompressors. This, in turn, lets applications or other software draw any QuickTime video directly to the video output component's graphics world.

This section gives an overview of developing this transfer codec. Bear in mind that a transfer codec is a specialized image decompressor component, and should be based on the Base Image Decompressor.

Overview of Transfer Codecs

QuickTime 2.5 contained new support for developers of codecs to accelerate certain image decompression operations. These features will most likely be used by developers of video hardware boards that provide special acceleration features, such as arbitrary scaling or color space conversion.

Prior to QuickTime 2.5, if a codec could not decompress an image directly to the screen, the ICM would prepare an offscreen buffer for the codec, then use the None codec to transfer the image from the offscreen buffer to the screen. With QuickTime 2.5, if a codec cannot decompress directly to the screen it has the option of specifying that it can decompress to one or more types of non-RGB pixel spaces, specified as an `OSType` (e.g., `'yuvs'`). The ICM then attempts to find a decompressor component of that type (a transfer codec) that

can transfer the image to the screen. Since the ICM does not define non-RGB pixel types, the transfer codec must support additional calls to set up the offscreen. If a transfer codec cannot be found that supports the specified non-RGB pixel types, the ICM uses the None codec with an RGB offscreen buffer.

The real speed benefit comes from the fact that since the transfer codec defines the offscreen buffer, it can place the buffer in on-board memory, or even point to an overlay plane so that the offscreen image really is on the screen. In this case, the additional step of transferring the bits from offscreen memory on to the screen is avoided.

Creating a Transfer Codec for a Video Output Component

For an image decompressor component to indicate that it can decompress to non-RGB pixel types, it should, in the `ImageCodecPreDecompress` call, fill in the `wantedDestinationPixelTypes` field with a handle to a zero-terminated list of pixel types that it can decompress to. The ICM immediately makes a copy of the handle. Cinepak, for example, returns a 12-byte handle containing `yuvs`, `yuvu`, and $00000000. Since `ImageCodecPreDecompress` can be called often, it is suggested that codecs allocate this handle when their component is opened and simply fill in the `wantedDestinationPixelTypes` field with this handle during `ImageCodecPreDecompress`. Components that use this method should be sure to dispose the handle at close.

Apple's Cinepak decompressor supports decompressing to `'yuvs'` and `'yuvu'` pixel types. Type `'yuvs'` is a YUV format with u and `v` components signed (center point at $00), while `'yuvu'` has the u and `v` component centered at $80.

As an example, suppose XYZ Co. had a video board that had a YUV overlay plane capable of doing arbitrary scaling. The overlay plane takes data in the same format as Cinepak's `'yuvs'` format. In this case, XYZ would make a component of type `'imdc'` and subtype `'yuvs'`.

The `ImageCodecPreDecompress` call would set the `codecCanScale`, `codecHasVolatileBuffer`, and `codecImageBufferIsOnScreen` bits in the `capabilities`.flags field. The `codecImageBufferIsOnScreen` bit is necessary to inform the ICM that the codec is a direct screen transfer codec. A direct screen transfer codec is one that sets up an offscreen buffer that is actually onscreen (such as an overlay plane). Not setting this bit correctly can cause unpredictable results.

The real work of the codec takes place in the `ImageCodecNewImageBufferMemory` call. This is where the codec is instructed to prepare the non-RGB pixel buffer. The codec must fill in the `baseAddr` and `rowBytes` fields of the `dstPixMap` structure in the `CodecDecompressParams`. The ICM then passes these values to the original codec (e.g., Cinepak) to decompress into.

The codec must also implement `ImageCodecDisposeMemory` to balance `ImageCodecNewImageBufferMemory`.

Since Cinepak then decompresses into the card's overlay plane, `ImageCodecBandDecompress` needs to do nothing aside from calling `ICMDecompressComplete`.

```
pascal ComponentResult ImageCodecPreDecompress( Handle storage,
    CodecDecompressParams *p)
{
    CodecCapabilities *capabilities = p->capabilities; // only allow 16 bpp
                                                       // source
    if ((**p->imageDescription).depth != 16)
        return codecConditionErr; /* we only support 16 bits per pixel dest */
    if (p->dstPixMap.pixelSize != 16)
```

```
        return codecConditionErr;

    capabilities->wantedPixelSize = p->dstPixMap.pixelSize;
    capabilities->bandInc = capabilities->bandMin =
                                        (*p->imageDescription)->height;
    capabilities->extendWidth = 0;
    capabilities->extendHeight = 0;
    capabilities->flags = codecCanScale | codecImageBufferIsOnScreen |
            codecHasVolatileBuffer;
    return noErr;
}

pascal ComponentResult ImageCodecBandDecompress(Handle storage,
    CodecDecompressParams *p)
{
    ICMDecompressComplete(p->sequenceID, noErr, codecCompletionSource |
        codecCompletionDest, &p->completionProcRecord);
    return noErr;
}

pascal ComponentResult ImageCodecNewImageBufferMemory(Handle storage,
    CodecDecompressParams *p, long flags,
    ICMMemoryDisposedUPP memoryGoneProc, void *refCon)
{
    OSErr err = noErr;
    long offsetH, offsetV;
    Ptr baseAddr;
    long rowBytes;

    // call predecompress to check to make sure we can handle
    // this destination
    err = ImageCodecPreDecompress(storage, p);
    if (err) goto bail;

    // set video board registers with the scale
    XYZVideoSetScale(p->matrix);

    // calculate a base address to write to
    offsetH = (p->dstRect.left - p->dstPixMap.bounds.left);
    offsetV = (p->dstRect.top - p->dstPixMap.bounds.top);
    XYZVideoGetBaseAddress(p->dstPixMap, offsetH, offsetV,
        &baseAddr, &rowBytes);

    p->dstPixMap.baseAddr = baseAddr;
    p->dstPixMap.rowBytes = rowBytes;
    p->capabilities->flags = codecImageBufferIsOnScreen;

    bail:
        return err;
}

pascal ComponentResult ImageCodecDisposeMemory(Handle storage, Ptr data)
{
    return noErr;
}
```

Some video hardware boards that use an overlay plane require that the image area on screen be flooded with a particular RGB value or alpha-channel in order to have the overlay buffer show through at that location. Codecs that require this support should set the `screenFloodMethod` and `screenFloodValue` fields of the `CodecDecompressParams` record during `ImageCodecPreDecompress`. The ICM then manages the flooding of the screen buffer. This method is more reliable than having the codec attempt to flood the screen itself, and will ensure compatibility with future versions of QuickTime.

Using Video Output Components

This section explains how to use video output components in your software.

Video output components are standard components that are managed by the Component Manager. Their component type is `QTVideoOutputComponentType`.

Apple has defined a functional interface for video output components. For each function of a video output component, there is a selector constant. These constants are listed in the next section.

Selectors for Video Output Component Functions

The following constants are the selectors for functions of a video output component.

```
enum {
    kQTVideoOutputGetDisplayModeListSelect = 0x0001,
    kQTVideoOutputGetCurrentClientNameSelect = 0x0002,
    kQTVideoOutputSetClientNameSelect = 0x0003,
    kQTVideoOutputGetClientNameSelect = 0x0004,
    kQTVideoOutputBeginSelect = 0x0005,
    kQTVideoOutputEndSelect = 0x0006,
    kQTVideoOutputSetDisplayModeSelect = 0x0007,
    kQTVideoOutputGetDisplayModeSelect = 0x0008,
    kQTVideoOutputCustomConfigureDisplaySelect = 0x0009,
    kQTVideoOutputSaveStateSelect = 0x000A,
    kQTVideoOutputRestoreStateSelect = 0x000B,
    kQTVideoOutputGetGWorldSelect = 0x000C,
    kQTVideoOutputGetGWorldParametersSelect = 0x000D,
    kQTVideoOutputGetIndSoundOutputSelect = 0x000E,
    kQTVideoOutputGetClockSelect = 0x000F,
    kQTVideoOutputSetEchoPortSelect = 0x0010
};
```

Selecting a Video Output Component

Listing 4-1 shows how to assemble a list of available video output components using the `FindNextComponent` function. This list can then be presented to the user.

The base video output component is a special component that provides services to other video output components. It is never connected to a display, and it has a component flag, `kQTVideoOutputDontDisplayToUser`, that indicates that it should not be included in a list of available video output components. The sample code shows how to check for this flag.

Listing 4-1 Displaying available video output components

```
ComponentDescription cd;
Component c = 0;
cd.componentType = QTVideoOutputComponentType;
cd.componentSubType = 0;
cd.componentManufacturer = 0;
cd.componentFlags = 0;
cd.componentFlagsMask = kQTVideoOutputDontDisplayToUser;
while (c = FindNextComponent (c, &cd)) {
    Handle nameHandle = NewHandle (0);
    GetComponentInfo (c, &cd, nameHandle, nil, nil);
    // add name to list
    DisposeHandle (nameHandle);
}
```

Choosing a Display Mode

After a video output component is chosen, the next step is to choose one of the component's available display modes. Your software does this by getting the QT atom container that contains descriptions of the available modes by calling the `QTVideoOutputGetDisplayModeList` function, traversing the atom container's contents using the `QTFindChildByIndex` function, examining the characteristics of each mode and setting aside any modes that are not appropriate for your software, and then optionally presenting a list of modes to the user to select from.

If your software does present a list of display modes to the user, it can obtain a string that describes each mode from the mode's `kQTVOName` atom with an ID of 1. The string doesn't include a leading length byte or a trailing null. Your software can determine the length of the string from the size of the atom.

When the mode has been chosen, your software calls the `QTVideoOutputSetDisplayMode` function to set the display mode.

Of display mode's characteristics, the most important is whether the mode can display the video data. This is determined by the availability of a decompressor component that takes the video data as input and converts it to the type of data, specified by the `kQTVOPixelType` atom, required by the video output device. If a video output device can directly display one of the supported QuickTime pixel formats, the necessary decompressor component is included in QuickTime. If special decompressor components are required for the video output device, such as JPEG or other decompressors that deliver data directly to the video output hardware without creating a new pixel format, these decompressor components are described in `kQTVODecompressors` atoms.

Contents of the Display Mode QT Atom Container

To obtain descriptions of the display modes, your software must traverse QT atom containers. At the root of the QT atom container returned by the `QTVideoOutputGetDisplayModeList` function are one or more atoms of type `kQTVODisplayModeItem`, each containing a definition of a display mode. Your software can traverse the display mode atoms by calling the `QTFindChildByIndex` function.

Within each `kQTVODisplayModeItem` atom are the following atoms:

- The atom of type `kQTVODimensions` with ID 1 contains two 32-bit integers. The first specifies the width, in pixels, of the display. The second specifies the height, in pixels, of the display.
- The atom of type `kQTVOResolution` with ID 1 contains two 32-bit fixed-point values. The first specifies the horizontal resolution of the display, in pixels per inch. The second specifies the vertical resolution of the display, in pixels per inch.

By storing resolutions rather than an aspect ratio, QuickTime makes it easy for your software to compare values with values in QuickTime `ImageDescription` records. Your software can calculate the aspect ratio for the display mode by dividing the value for the horizontal resolution by the value for the vertical resolution.

- The atom of type `kQTVORefreshRate` with ID 1 contains a single 32-bit fixed-point value. This value specifies the refresh rate of the display in Hertz.
- The atom of type `kQTVOPixelType` with ID 1 contains a single 32-bit `OSType` value. This value specifies the type of pixel that is used by the display format:
- Values of 1, 2, 4, 8, 16, 24 and 32 specify standard Mac OS RGB pixel formats with corresponding bit depths.
- Values of 33, 34, 36 and 40 specify standard Mac OS gray-scale pixel formats with depths of 1, 2, 4, and 8 bits per pixel.
- Other pixel formats are specified by four-character codes. There are currently codes for RGB pixel formats defined for Microsoft Windows and for several YUV formats. For information about pixel formats defined for Microsoft Windows, see *QuickTime Guide for Windows*.
- The atom of type `kQTVOName` with ID 1 contains a string that describes the display mode. Your software can use this string when presenting a list of available display modes to the user. The string does not include a leading length byte or a trailing null. Your software can determine the length of the string by getting the size of the atom that contains it.
- Atoms of type `kQTVODecompressors` specify any special decompressors that are required for the video output device. If a video output device cannot directly display 32-bit RGB data or data in one of the other supported QuickTime pixel formats, a special decompressor is required to convert images to data that the video output device can display.

Because `kQTVODecompressors` atoms are not required to have consecutive IDs, your software must use the `QTFindChildByIndex` function to iterate through the decompressors.

Within each `kQTVODecompressors` atom are one or more atoms:

- The atom of type `kQTVODecompressorType` with ID 1 contains an `OSType` value that specifies the type of compressed data that the decompressor can decompress. For example, a `kQTVODecompressorType` atom that contains `kMotionJPEGACodecType` can decompress Motion JPEG Format A data.
- An atom of type `kQTVODecompressorComponent` with ID 1 is optional. If present, it contains a `DecompressorComponent` value that specifies a decompressor component that your software can use to decompress the data specified by the corresponding `kQTVODecompressorType` atom.
- An atom of type `kQTVODecompressorContinuous` with ID 1 is optional. If present, it contains a Boolean value that specifies whether the resulting video display will be continuous. If the value is `true`, data will be displayed without any visual gaps between successive images. If the value is `false`, data will be displayed, but there may be a visual gap (such as a black screen) between the display of images. If there is no `kQTVODecompressorContinuous` atom, your software should not make any assumptions about the performance of the decompressor.

Drawing to an Echo Port

Some video output devices can display video simultaneously on an external video display and in a window on a computer's desktop. To use this feature, your software draws to a graphics port for the window on the computer's desktop, known as the echo port, rather than the port that is normally used for the video output device. The video then appears on both displays, although in some cases the video on the desktop is displayed at a smaller size or lower frame rate.

To draw to both outputs at the same time, do the following:

- Call the `ComponentFunctionImplemented` function to determine if the video output component supports the `QTVideoOutputSetEchoPort` function.
- Call the `QTVideoOutputSetEchoPort` function to specify a window on the desktop in which to display video sent to the device.
- Call the `SetMovieGWorld` function to specify the same window for the output of a movie.

This process is shown in Listing 4-2.

Listing 4-2 Drawing to an echo port

```
Movie               aMovie;
ComponentInstance   ci;
CGrafPtr            thePort;
/* instantiation of the video output component here */
/* creation of the graphics port here */
if (ComponentFunctionImplemented(ci, kQTVideoOutputSetEchoPortSelect)) {
    result = QTVideoOutputSetEchoPort(ci, thePort);
    SetMovieGWorld (aMovie, thePort, nil);
    StartMovie (aMovie);
}
```

The video then appears on both displays. Note that you bypass the graphics world that is normally used for the video output device; your software draws only to the echo port you specify with the `QTVideoOutputSetEchoPort` function.

Functions Used To Control Video Output Components

This section discusses the functions used to control video output components.

Controlling the Display Mode

Each video output device has a finite number of display modes. Each mode has several characteristics, including width and height of the display, pixel depth, and video refresh rate. This section describes functions for getting and setting the display mode.

To get a list of the display modes supported by a video output component, call the `QTVideoOutputGetDisplayModeList` function. The list is a QT atom container, and list atoms contain the characteristics of each mode. You use QT atom container functions, such as `QTFindChildByIndex`, to extract the contents of the list.

To specify a display mode to use, call the `QTVideoOutputSetDisplayMode` function.

To find out the current display mode, call the `QTVideoOutputGetDisplayMode` function.

Registering the Name of Your Software

After your software has established a connection to a video output component, you can register its name with the instance of that component by calling the `QTVideoOutputSetClientName` function. The name can then be used by `QTVideoOutputGetCurrentClientName` to specify which software has exclusive access to the video output device controlled by the component.

Although several applications or other software can connect to a video output component at the same time, only one of them at a time can have access to the video output device controlled by the component. Use `QTVideoOutputBegin` to gain exclusive access to the video output device and `QTVideoOutputEnd` to relinquish exclusive access when your software has finished using the device.

To get the name of the application or other software that is registered with an instance of a video output component, call `QTVideoOutputGetClientName`.

Controlling Video Output

Video output components provide functions for configuring the video display, for starting and stopping video output, and for specifying the graphics world used for the display:

- To display a dialog box in which the user can specify video settings, call the `QTVideoOutputGetConfigureDisplay` function.
- To get a pointer to the graphics world used by the video output component, call the `QTVideoOutputGetGWorld` function.
- To obtain exclusive access to the video hardware controlled by a video output component, call the `QTVideoOutputBegin` function.
- To release access to the video hardware controlled by a video output component, call the `QTVideoOutputEnd` function.
- To get the name of the software, if any, that has exclusive access to the video hardware controlled by a video output component, call the `QTVideoOutputGetCurrentClientName` function.
- If a video output device can display video both on an external video display and in a window on a computer's desktop, you can use the `QTVideoOutputSetEchoPort` function to specify a window on the desktop in which to display video sent to the device.

Finding Associated Components

Video output components provide functions for finding other components associated with them:

- To find any sound output components associated with a video output component, call the `QTVideoOutputGetIndSoundOutput` function.
- To find a clock component associated with a video output component, call the `QTVideoOutputGetClock` function.

Saving and Restoring Component Configurations

Video output components provide functions for saving the current configuration of a video output component and later restoring the configuration:

- To save the current configuration of a video output component, call the `QTVideoOutputSaveState` function.
- To restore a previously saved configuration of a video output component, call the `QTVideoOutputRestoreState` function.

Data Types

This section describes the QT atom container used to specify the display modes that are supported by a video display component.

Display Mode QT Atom Container

The `QTVideoOutputGetDisplayModeList` function returns a list of the display modes supported by a video display component. This list is contained in the QT atom container described in this section.

At the root of the QT atom container returned by the `QTVideoOutputGetDisplayModeList` function are one or more atoms of type `kQTVODisplayModeItem`, each containing a definition of a display mode. Your software can traverse the display mode atoms by calling the `QTFindChildByIndex` function.

Within each `kQTVODisplayModeItem` atom are the following atoms:

- The atom of type `kQTVODimensions` with ID 1 contains two 32-bit integers. The first specifies the width, in pixels, of the display. The second specifies the height, in pixels, of the display.
- The atom of type `kQTVOResolution` with ID 1 contains two 32-bit fixed-point values. The first specifies the horizontal resolution of the display, in pixels per inch. The second specifies the vertical resolution of the display, in pixels per inch.

By storing resolutions rather than an aspect ratio, QuickTime makes it easy for your software to compare values with values in QuickTime `ImageDescription` records. Your software can calculate the aspect ratio for the display mode by dividing the value for the horizontal resolution by the value for the vertical resolution.

- The atom of type `kQTVORefreshRate` with ID 1 contains a single 32-bit fixed-point value. This value specifies the refresh rate of the display in Hertz.
- The atom of type `kQTVOPixelType` with ID 1 contains a single 32-bit `OSType` value. This value specifies the type of pixel that is used by the display format:
- Values of 1, 2, 4, 8, 16, 24 and 32 specify standard Mac OS RGB pixel formats with corresponding bit depths.
- Values of 33, 34, 36 and 40 specify standard Mac OS gray-scale pixel formats with depths of 1, 2, 4, and 8 bits per pixel.
- Other pixel formats are specified by four-character codes. There are currently codes for RGB pixel formats defined for Microsoft Windows and for several YUV formats.
- The atom of type `kQTVOName` with ID 1 contains a string that describes the display mode. Your software can use this string when presenting a list of available display modes to the user. The string does not include a leading length byte or a trailing null. Your software can determine the length of the string by getting the size of the atom that contains it.
- Atoms of type `kQTVODecompressors` specify any special decompressors that are required for the video output device. If a video output device cannot directly display 32-bit RGB data or data in one of the other supported QuickTime pixel formats, a special decompressor is required to convert images to data that the video output device can display.

Because `kQTVODecompressors` atoms are not required to have consecutive IDs, your software must use the `QTFindChildByIndex` function to iterate through the decompressors.

Within each `kQTVODecompressors` atom are one or more atoms:

- The atom of type `kQTVODecompressorType` with ID 1 contains an `OSType` value that specifies the type of compressed data that the decompressor can decompress. For example, a `kQTVODecompressorType` atom that contains `kMotionJPEGACodecType` can decompress Motion JPEG Format A data.

- An atom of type `kQTVODecompressorComponent` with ID 1 is optional. If present, it contains a `DecompressorComponent` value that specifies a decompressor component that your software can use to decompress the data specified by the corresponding `kQTVODecompressorType` atom.
- An atom of type `kQTVODecompressorContinuous` with ID 1 is optional. If present, it contains a Boolean value that specifies whether the resulting video display will be continuous. If the value is `true`, data will be displayed without any visual gaps between successive images. If the value is `false`, data will be displayed, but there may be a visual gap (such as a black screen) between the display of images. If there is no `kQTVODecompressorContinuous` atom, your software should not make any assumptions about the performance of the decompressor.

Constants

This section provides details on component type, atom type, and function selector constants.

Component Instance, Type, and Subtype

```
typedef ComponentInstance QTVideoOutputComponent;
enum {
    QTVideoOutputComponentType = FOUR_CHAR_CODE('vout'),
    QTVideoOutputComponentBaseSubType = FOUR_CHAR_CODE('base')
};
```

Video Output Component Flag

The following flag indicates that a video output component is not connected to a display and should not be included in a list of components that are available to the user.

```
enum {
    kQTVideoOutputDontDisplayToUser = 1L << 0
};
```

Display Mode Atom Types

The following atom type constants specify atom types:

```
enum {
    kQTVODisplayModeItem = FOUR_CHAR_CODE('qdmi'),
    kQTVODimensions = FOUR_CHAR_CODE('dimn'),
        /* atom contains two longs - pixel count - width, height */
    kQTVOResolution = FOUR_CHAR_CODE('resl'),
        /* atom contains two Fixed - hRes, vRes in dpi */
    kQTVORefreshRate = FOUR_CHAR_CODE('refr'),
        /* atom contains one Fixed - refresh rate in Hz */
    kQTVOPixelType = FOUR_CHAR_CODE('pixl'),
        /* atom contains one OSType - pixel format of mode */
    kQTVOName = FOUR_CHAR_CODE('name'),
        /* atom contains string (no length byte) --
```

```
            name of mode for display to user */
    kQTVODecompressors = FOUR_CHAR_CODE('deco'),
        /* atom contains other atoms indicating supported decompressors */
        /* kQTVODecompressors sub-atoms */
    kQTVODecompressorType = FOUR_CHAR_CODE('dety'),
        /* atom contains one OSType - decompressor type code */
    kQTVODecompressorContinuous = FOUR_CHAR_CODE('cont'),
        /* atom contains one Boolean --
           true if this type is displayed continuously */
    kQTVODecompressorComponent = FOUR_CHAR_CODE('cmpt')
        /* atom contains one component id of decompressor */
};
```

Creating Video Output Components

This section describes the routines a hardware developer must implement when creating a video output component.

The examples in this section show how your video output component can use the services of the base video output component provided by Apple Computer. If your component uses these services, you do not have to implement the entire API for a video output component. You simply implement the functions described here, and the base video output component handles the rest. For most of the functions, you extend functions already included in the base video output component, which is much faster than providing complete implementations of these functions yourself. If the base video output component's implementation of any of these functions returns an error, the function in your video output component must immediately return with the same error. If the base video output component's implementation completes successfully, then your video output component's function provides the remainder of the implementation.

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

Connecting to the Base Video Output Component

To use the services of the base video output component, your video output component must open a connection to the base video output component. It does this in its routine for processing open requests from the Component Manager. How to connect to the base video output component is shown in Listing 6-1.

Listing 6-1 Connecting to the base video output component

```
QTVideoOutputComponent baseVideoOutput;
OSErr err;
err = OpenADefaultComponent (kVideoOutputComponentType,
                             kVideoOutputComponentBaseSubType,
                             &baseVideoOutput);
err = ComponentSetTarget (baseVideoOutput,
                          self);
globals->baseVideoOutput = baseVideoOutput;
```

Providing a Display Mode List

Your video output component must implement its own `QTVideoOutputGetDisplayModeList` function. This function is required for all video output components.

Starting Video Output

Listing 6-2 shows how your video output component can start video output.

Listing 6-2 Starting video output

```
pascal ComponentResult MyQTVideoOutputBegin (Globals storage)
{
    ComponentResult err;
    long mode;
    // call the default implementation
    err = QTVideoOutputBegin (storage->baseVideoOutput);
    if (err) goto bail;
    // get the selected mode
    err = QTVideoOutputGetDisplayMode (storage->self, &mode);
    if (err) goto bail;
    // switch the hardware to the selected mode
    // remember that we now own the hardware
    storage->ownHardware = true;
bail:
    if ((err != noErr) && (storage->ownHardware == true))
        QTVideoOutputEnd (storage-> baseVideoOutput);
    return err;
}
```

The default implementation of the `QTVideoOutputBegin` function ensures that the hardware is not currently in use by other software. It also ensures that a valid display mode has been set with either the `QTVideoOutputSetDisplayMode` function or the `QTVideoRestoreSettings` function.

Ending Video Output

Listing 6-3 shows how your video output component can stop video output. The implementation of this function is similar to the implementation of `QTVideoOutputEnd`, but here the default implementation must be called after the hardware has been released.

Listing 6-3 Ending video output

```
pascal ComponentResult MyQTVideoOutputEnd (Globals storage)
{
    ComponentResult err;
    // check that Begin has been called
    if (storage->ownHardware == false) {
        err = paramErr;
        goto bail;
    }
    // release the hardware
    // call default implementation
    QTVideoOutputEnd (storage->baseVideoOutput);
    // remember that we no longer own the hardware
    store->ownHardware = false;
bail:
    return err;
```

```
}
```

In the implementation of `QTVideoOutputEnd`, your component should also display a default image on the video output device to indicate that the device is no longer in use by other software.

Implementing the QTVideoOutputSaveState Function

If your video output component uses any custom settings, your component must implement its own `QTVideoOutputSaveState` function to save them. If your video output component has no custom settings, it can use the default `QTVideoOutputSaveState` implementation provided by the base video output component. Listing 6-4 shows an implementation of the `QTVideoOutputSaveState` function that saves custom settings. The function creates a QT atom container for storing the settings.

Listing 6-4 Extending the QTVideoOutputSaveState function

```
pascal ComponentResult MyQTVideoOutputSaveState (Globals storage,
                                        QTAtomContainer *settings)
{
    OSErr err;
    // call default implementation
    err = QTVideoOutputSaveState (storage->baseVideoOutput, settings);
    if (err) goto bail;

    // add custom parameter(s)
    err = QTInsertChild (*settings,kParentAtomIsContainer,
                         'FOOB', 1, 0,
                         sizeof (storage->customSetting),
                         &storage->customSetting, nil);
    if (err) goto bail;
bail:
    return err;
}
```

Implementing the QTVideoOutputRestoreState Function

If your video output component saves custom settings with its own implementation of the `QTVideoOutputSaveState` function, it must also implement a `QTVideoOutputRestoreState` function to restore the settings. If your video output component has no custom settings, it can use the default `QTVideoOutputRestoreState` implementation provided by the base video output component. Listing 6-5 shows an implementation of the `QTVideoOutputRestoreState` function that restores custom settings from the QT atom container in which they are stored.

Listing 6-5 Restoring custom settings

```
pascal ComponentResult MyQTVideoOutputRestoreState (Globals storage,
                                        QTAtomContainer settings)
{
    OSErr err;
    QTAtom atom;
    // call default implementation
```

```
    err = QTVideoOutputRestoreState (storage->baseVideoOutput, settings);
    if (err) goto bail;
    // get custom parameter(s)
    atom = QTFindChildByID (settings, kParentAtomIsContainer, 'FOOB',
                            1, nil);
    if (atom != 0) {
        long dataSize;
        Ptr dataptr;
        QTGetAtomDataPtr (settings, atom, &dataSize, &dataPtr);
        storage->customSetting = *(SettingsType *)dataPtr;
    }
    else {
        // reset custom settings to default values
    }
bail:
    return err;
}
```

Implementing the QTVideoOutputGetGWorldParameters Function

Your video output component must also implement the `QTVideoOutputGetGWorldParameters`. This function is not called by applications or other clients of your component; it is called by the base video output component as part of the implementation of the `QTVideoOutputGetGWorld` function.

```
pascal ComponentResult QTVideoOutputGetGWorldParameters (
    QTVideoOutputComponent vo,
    Ptr *baseAddr,
    long *rowBytes,
    CTabHandle *colorTable);
```

In the `baseAddr` parameter, your video output component must return the address at which to display pixels. If your component does not display pixels, return 0 for this parameter.

In the `rowBytes` parameter, your video output component must return the width of each scan line in bytes. If your component does not display pixels, return the width of the current display mode.

In the `colorTable` parameter, your video output component must return the color table to be used. If your component does not use a color table, return `nil`.

Controlling Other Hardware Features

If the video output device includes features that can be controlled by any of the following functions, the video output component must implement the functions for those features.

- `QTVideoOutputGetIndSoundOutput`
- `QTVideoOutputGetIndImageDecompressor`
- `QTVideoOutputGetClock`
- `QTVideoOutputCustomConfigureDisplay`

- `QTVideoOutputSetEchoPort`

Delegating Other Component Calls

Your video output component's dispatcher must delegate all component selectors it doesn't handle itself to the base video output component. It can do this by calling the `DelegateComponentCall` function.

Closing the Connection to the Base Video Output Component

When your video output component closes, it must close its connection to the base video output component by calling the `CloseComponent` function.

Creating Data Handler Components

This section describes the requirements for creating a data handler component. The functional interface that your component must support is described in Using Data Handler Components (page 11).

You should consider developing your own data handler component only if you are planning to provide a new type of movie container or a container that requires special data handling techniques. For example, if you are planning to develop a networked multimedia server, you would most likely need to develop a new data handler that could support the special protocols required by your server. By encapsulating that protocol support in a data handler, QuickTime applications can access the movie data on your server without having to implement any special support. In this way, your server becomes a seamless part of the user's system.

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

General Information

All data handler components have a component type value of `'dhlr'`, which is defined by the `dataHandlerType` constant. Data handler components use the value of the component subtype field to indicate the type of data reference they support. As a result of this convention, note that all data handlers that share a component subtype value must be able to recognize and work with data references of the same type. For example, file system data handlers always carry a component subtype value of `'alis'`, which indicates that their data references are file system aliases (note that this is true for QuickTime on Macintosh and under Windows, even though there is not, properly, a file system alias under Windows). Apple's memory-based data handler for Macintosh has a component subtype value of `'hndl'`.

```
#define dataHandlerType 'dhlr'
#define rAliasType      'alis'
```

Apple has not defined any special manufacturer field values or component flags values for data handler components. Developers may use the manufacturer field value to select your data handler from among all the data handlers that support a given type of data reference.

Apple has defined a functional interface for data handler components. You can use the following constants to refer to the request codes for each of the functions that your component must support:

```
enum {
    kDataHGetDataSelect        = 2,    /* DataHGetData */
    kDataHPutDataSelect        = 3,    /* DataHPutData */
    kDataHFlushDataSelect      = 4,    /* DataHFlushData */
    kDataHOpenForWriteSelect   = 5,    /* DataHOpenForWrite */
    kDataHCloseForWriteSelect  = 6,    /* DataHCloseForWrite */
    kDataHOpenForReadSelect    = 8,    /* DataHOpenForRead */
    kDataHCloseForReadSelect   = 9,    /* DataHCloseForRead */
    kDataHSetDatRefSelect      = 10,   /* DataHSetDataRef */
    kDataHGetDataRefSelect     = 11,   /* DataHGetDataRef */
    kDataHCompareDataRefSelect = 12,   /* DataHCompareDataRef */
    kDataHTaskSelect           = 13,   /* DataHTask */
```

```
    kDataHScheduleDataSelect   = 14,   /* DataHScheduleData */
    kDataHFinishDataSelect     = 15,   /* DataHFinishData */
    kDataHFlushCacheSelect     = 16,   /* DataHFlushCache */
    kDataHResolveDataRefSelect = 17,   /* DataHResolveDataRef */
    kDataHGetFileSizeSelect    = 18,   /* DataHGetFileSize */
    kDataHCanUseDataRefSelect  = 19,   /* DataHCanUseDataRef */
    kDataHGetVoumeListSelect   = 20,   /* DataHGetVolumeList */
    kDataHWriteSelect=         = 21,   /* DataHWrite */
    kDataHPreextendSelect      = 22,   /* DataHPreextend */
    kDataHSetFileSizeSelect    = 23,   /* DataHSetFileSize */
    kDataHGetFreeSpaceSelect   = 24,   /* DataHGetFreeSpace */
    kDataHCreateFileSelect     = 25,   /* DataHCreateFile */
    kDataHGetPreferredBlockSizeSelect = 26,
                                       /*DataHGetPreferredBlockSize */
    kDataHGetDeviceIndexSelect = 27,   /* DataHGetDeviceIndex */
    /* 28 and 29 unused */
    kDataHGetScheduleAheadTimeSelect= 30,
                                       /* DataHGetScheduleAheadTime */
    kDataHSetOSFileRefSelect   = 516,  /* DataHSetOSFileRef */
    kDataHGetOSFileRefSelect   = 517,  /* DataHGetOSFileRef */
    kDataHPlaybackHintsSelect  = 3+0x100/* DataHPlaybackHints */
};
```

Supported Functions

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

- Selecting a Data Handler (page 11)
  - `DataHGetVolumeList`
  - `DataHCanUseDataRef`
  - `DataHGetDeviceIndex`
- Working With Data References (page 13)
  - `DataHSetDataRef`
  - `DataHGetDataRef`
  - `DataHCompareDataRef`
  - `DataHResolveDataRef`
  - `DataHSetOSFileRef`
  - `DataHGetOSFileRef`
- Reading Movie Data (page 14)
  - `DataHOpenForRead`
  - `DataHCloseForRead`
  - `DataHGetData`
  - `DataHScheduleData`

- DataHFinishData
- DataHGetScheduleAheadTime

- Writing Movie Data (page 15)
  - DataHOpenForWrite
  - DataHCloseForWrite
  - DataHPutData
  - DataHWrite
  - DataHSetFileSize
  - DataHGetFileSize
  - DataHCreateFile
  - DataHGetPreferredBlockSize
  - DataHGetFreeSpace
  - DataHPreextend

- Managing Data Handler Behavior (page 16)
  - DataHTask
  - DataHFlushCache
  - DataHFlushData
  - DataHPlaybackHints

- Completion Function (page 14)

Document Revision History

This table describes the changes to *QuickTime Transport and Delivery Guide*.

Date	Notes
2006-01-10	New document that describes components that transport data between QuickTime movies and specific devices.
	Replaces "Data Handler Components" and "Video Output Components."
2002-09-17	New document that explains how to create a QuickTime data handler or sequence grabber component.

Document Revision History