Parts

objVideoWindow
    Object expression that evaluates to an IVideoWindow object.
handle
    New value for the window handle.

Remarks

This property offers a way for applications to set the owner of the video window. This is often used when playing videos in compound documents.

Previous Home Topic Contents Index Next

SetWindowForeground Method (IVideoWindow Object)

IVideoWindow Object

Sets the video window as the foreground window and optionally gives it focus.

objVideoWindow.**SetWindowForeground** *Focus*

Parts

objVideoWindow
    Object expression that evaluates to an IVideoWindow object.
Focus
    Long value that specifies whether the video window will have focus. A value of −1 gives the window focus and 0 does not.

Previous Home Topic Contents Index Next

SetWindowPosition Method (IVideoWindow Object)

IVideoWindow Object

Sets the position of the video window (not the client rectangle position) in device coordinates.

objVideoWindow.**SetWindowPosition** *Left*, *Top*, *Width*, *Height*

Parts

objVideoWindow
Object expression that evaluates to an IVideoWindow object.
Left
Specifies the x-axis origin of the window.
Top
Specifies the y-axis origin of the window.
Width
Specifies the width of the window.
Height
Specifies the height of the window.

Remarks

Specify, in window coordinates, where the video should appear. For example, setting a destination of (100,50,200,400) positions the video playback at an origin of 100 pixels from the left of the client area, 50 pixels from the top, and with an overall size of 200 x 400 pixels. If the video is smaller than this (or a source rectangle has been specified that is smaller than the video), it will be stretched appropriately. Similarly, if the video is larger than the destination rectangle, the video is compressed into the visible rectangle. There are fairly severe performance penalties if an application does not keep the source and destination rectangles the same size.

Under typical circumstances, when no destination rectangle has been set, the video fills the entire visible client window area (regardless of how much the user has stretched the window). Also, the destination rectangle properties correctly return the size of the video window client area.

This method has the same effect as individually setting the Left, Top, Width, and Height properties.

Top Property (IVideoWindow Object)

IVideoWindow Object

Retrieves or sets the y-axis coordinate of the video window.

objVideoWindow.**Top** [= *lValue*]

Parts

objVideoWindow
  Object expression that evaluates to an IVideoWindow object.
lValue
  New value for the y-axis origin.

Remarks

Calling this method does not affect the height of the video window.

Visible Property (IVideoWindow Object)

IVideoWindow Object

Retrieves or sets the visibility of the video window.

objVideoWindow.**Visible** [= *boolean*]

Parts

objVideoWindow
  Object expression that evaluates to an IVideoWindow object.
boolean
  If set to True, the window is shown; if False, the window is hidden.

Width Property (IVideoWindow Object)

IVideoWindow Object

Retrieves or sets the width of the video window.

objVideoWindow.**Width** [= *lValue*]

Parts

objVideoWindow
 Object expression that evaluates to an IVideoWindow object.
lValue
 New value of the width.

Remarks

The **Width** property is independent of the video window's Height property (the x-coordinate).

WindowState Property (IVideoWindow Object)

IVideoWindow Object

Returns or sets the state of the video window.

objVideoWindow.**WindowState** [= *lValue*]

Parts

objVideoWindow
 Object expression that evaluates to an IVideoWindow object.
lValue
 New value for the **WindowState** property.

WindowStyle Property (IVideoWindow Object)

IVideoWindow Object

Retrieves or sets the style parameters for the video window.

objVideoWindow.**WindowStyle** [= *lValue*]

Parts

objVideoWindow
: Object expression that evaluates to an IVideoWindow object.

lValue
: New value for the **WindowStyle** property.

Remarks

For a complete list of window styles, see the CreateWindow function in the Microsoft® Platform Software Development Kit (SDK).

WindowStyleEx Property (IVideoWindow Object)

IVideoWindow Object

Changes the style parameters for the video window.

objVideoWindow.**WindowStyleEx** [= *lValue*]

Parts

objVideoWindow
: Object expression that evaluates to an IVideoWindow object.

lValue
: New value for the flags. Valid values include only those flags that can be set by the GWL_STYLE value of the Microsoft Win32 GetWindowLong function.

DirectShow Basics

This section contains articles covering basic DirectShow concepts, such as filter graph architecture and data flow, how to use the Filter Graph Editor tool, and a list of the filters and sample filters supplied with DirectShow. You can use this section as a high-level introduction to DirectShow. You need only a general understanding of programming and media to understand the topics in this section.

Using DirectShow

Microsoft® DirectShow™ is an architecture that controls and processes streams of multimedia data through custom or built-in filters. You can also use the set of media streaming interfaces to stream media data without creating filters. See Use Multimedia Streaming in DirectShow Applications for more information.

In addition to the architecture and the set of classes and interfaces to support it, DirectShow is also a run time that uses this architecture to enable users to play digital movies and sound encoded in various formats, including MPEG, AVI, MOV (Apple® QuickTime®) and WAV-formatted files. The DirectShow run time is a control (.ocx), called the ActiveMovie Control, and a set of dynamic-link libraries (DLLs) that enable you to play back supported media files.

DirectShow playback uses video and audio hardware cards that support the Microsoft DirectX® set of application programming interfaces (APIs). The video and audio capture capability lets you programmatically control your system's video and audio capture hardware, as well as video and audio compressors and decompressors (codecs). The Plug and Play capability lets DirectShow automatically retrieve and use your filters, once you register their properties.

Use the DirectShow architecture for most new multimedia applications for Windows® 95 or Windows NT®. With a few exceptions, it replaces multimedia playback services, APIs, and architectures provided by Microsoft in earlier versions of the Windows Software Development Kit (SDK). However, libraries will continue to be available and supported for applications that use the earlier Microsoft multimedia playback services, such as Microsoft Video for Windows.

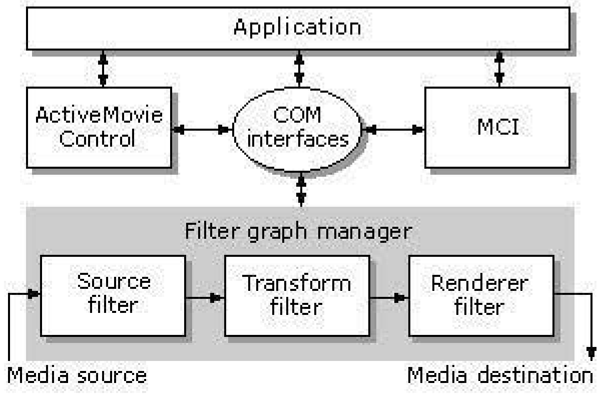
Contents of this article:

DirectShow Architecture

The DirectShow architecture defines how to control and process streams of multimedia data using modular components called *filters*. The filters have input or output pins, or both, and are connected to each other in a configuration called a *filter graph*. Applications use an object called the *filter graph manager* to assemble the filter graph and move data through it. By default, the filter graph manager automatically handles data flow for you; for example, it automatically inserts the proper codec if needed, and it automatically connects a transform filter's output pin to the default rendering filter. You can always specify your own filters and connections if you don't want to use the default configuration.

The filter graph manager provides a set of Component Object Model (COM) interfaces so that applications can access the filter graph. Applications can directly call the filter graph manager interfaces to control the media stream or retrieve filter events, or they can use the ActiveMovie Control to play back media files.

Thus, you can access DirectShow through the COM interface, the ActiveMovie Control, or media control interfaces (MCI), as shown in the following illustration.



Because of the DirectShow architecture's flexible, modular design, filter graphs have many potential uses and applications. Examples include filter graphs that implement video capture, control remote devices such as VCRs, or enable MIDI recording and editing.

Choosing the Right Programming Model

DirectShow is accessible at several levels, and the approach you use depends on what you need and how much programming you want to do. You might plan to rewrite an existing multimedia program, write a new multimedia program, or add multimedia capabilities to an existing program. Typically, existing applications that use the MCI command set are easily ported, whereas applications that access lower-level multimedia services require more time to rewrite. You can quickly add DirectShow playback services to new applications by using the ActiveMovie Control, or with a few direct functions that call the COM interfaces. C or C++ programmers can write filters that change or enhance multimedia data already managed by existing filter graphs.

This section contains the following topics.

- Rewriting Existing Applications
- Writing New Applications

Rewriting Existing Applications

If you have an application that plays AVI-encoded movies and sounds and want to adapt it to use DirectShow to play AVI files, porting is straightforward if your application uses MCI commands or the Microsoft Video for Windows® API. Your choice depends on the services the application uses and your goals. If your application uses MCI commands, you can use the MCI subset that DirectShow provides. In the majority of cases, this will be a straightforward upgrade that maintains AVI playback and adds MPEG and QuickTime playback capabilities to your application. If your existing C-based application uses Video for Windows API, you can replace most of these with calls to the COM interfaces.

Writing New Applications

You can take a variety of approaches when writing a new application with DirectShow. For example, if you only want to add MPEG playback to your application, you can incorporate the ActiveMovie Control into your application or directly access the COM interfaces on the filter graph manager. Both Microsoft Visual Basic® version 5.*x* and later and Microsoft Visual C++® version 5.*x* and later allow access to the ActiveMovie Control or the COM interfaces. Filters within a filter graph are typically written in C++ using the DirectShow class library.

If your application must process the media stream in some way or capture a media stream, you can incorporate both the filter graph manager and a custom filter into your application. The instantiated filter graph manager generates and manages the filter graph. You can insert the custom filter into a preconfigured filter graph (which you create and save by using the Filter Graph Editor tool in the DirectShow SDK). You also could insert the filter into an existing filter graph at run time.

Playing Back Files Over the Internet

The ActiveMovie Control is incorporated into Microsoft Internet Explorer so that you can place the control on a Web page and program it by using Microsoft Visual Basic® Scripting Edition (VBScript) commands. To a programmer, the ActiveMovie Control is another ActiveX™ Control, one that has real-time playback capability. Real-time playback means that the ActiveMovie Control can play video or audio files over the Internet while the file is downloading, rather than requiring the user to wait until the whole file is downloaded to begin playback.

The same filter graphs constructed to play media from files can play media from the Internet by simply changing the source filter. Take, for example, a filter graph that plays MPEG movies from a disk file. The first filter in the graph might be a file reader filter. By replacing this filter with a filter capable of reading from an Internet URL address, you can play MPEG movies from the Internet. Both file and URL reader source filters just deliver an unparsed stream of data. A parser filter pulls the data from the reader, parses it into separate streams of video, audio, text, or other data types, and pushes it downstream. This filter remains unchanged regardless of whether the source filter is a file or URL reader filter.

The source filter that reads from an Internet server is called the File Source (URL) filter. DirectShow provides this as a built-in filter. It knows how to read, but not parse, data from a URL address. Therefore, a media parser follows the File Source (URL) filter in the filter graph. For MPEG sources, this parser is built into the MPEG splitter filter. Other media types have their own parser filters (for example, a QuickTime parser).

The source filter that reads from files is the File Source (Async) filter. DirectShow also provides this as a built-in filter. It does no parsing on its own but simply reads data off a disk to play back. Most DirectShow filter graphs use this source filter.

The architecture's modularity allows most of the same components to be reused between file and Internet playback. This modularity also means that if you want to render a new type of data, often you only need to write a parser and renderer, and you can still use the existing file or URL filter.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

Filter Graph Manager and Filter Graphs

To use the filter graph manager from an application, it is not necessary to know much about the underlying filter graphs. However, it is useful to understand at least the basic principles of filter graphs if you ever want to configure your own filter graph rather than letting the filter graph manager configure them for you.

A filter graph is composed of a collection of filters of different types. Most filters can be categorized into one of the following three types.

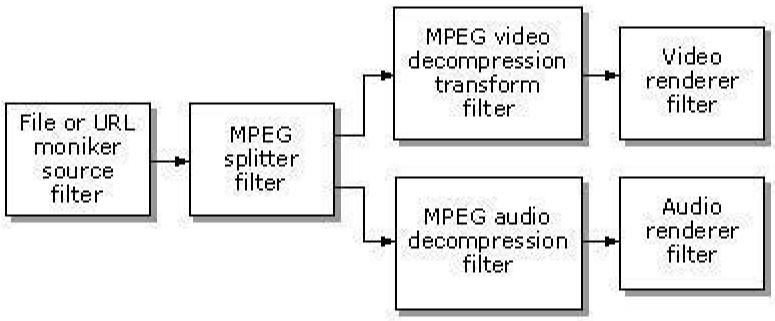
- A source filter, which takes the data from some source, such as a file on disk, a satellite feed, an Internet server, or a VCR, and introduces it into the filter graph.
- A transform filter, which takes the data, processes it, and then passes it along.
- A rendering filter, which renders the data; typically this is rendered to a hardware device, but could be rendered to any location that accepts media input (such as memory or a disk file).

In addition to these three types, there are other kinds of filters, for example, effect filters, which add effects without changing the data type, and parser filters, which understand the format of the source data and know how to read the correct bytes, create times stamps, and perform seeks.

For example, a filter graph whose purpose is to play back an MPEG-compressed video from a file would use the following filters.

- A source filter to read the data off the disk.
- An MPEG filter to parse the stream and split the MPEG audio and video data streams.
- A transform filter to decompress the video data.
- A transform filter to decompress the audio data.
- A video renderer filter to display the video data on the screen.
- An audio renderer filter to send the audio to the sound card.

The following illustration shows such a filter graph.


File or URL moniker source filter
MPEG splitter filter
MPEG video decompression transform filter
Video renderer filter
MPEG audio decompression filter
Audio renderer filter

It is possible for some filters to represent a combination of types. For example, a filter might be an audio renderer that also acts as a transform filter by passing through the video data. But typically, filters fit only one of these three types.

Filter graphs stream multimedia data through filters. In the media stream, one filter passes the media *downstream* to the next filter. An *upstream filter* describes the filter that passes data to the filter; a *downstream filter* describes the next filter in line for the data. This distinction is important because media flows downstream, but other information can go upstream.

To make a filter graph work, filters must be connected in the proper order, and the data stream must be started and stopped in the proper order. The filter graph manager connects filters and controls the media stream. It also has the ability to search for a configuration of filters that will render a particular media type and build that filter graph. Filter graphs can also be preconfigured, in which case the filter graph manager does not need to search for a configuration.

When searching for a rendering configuration, the filter graph manager uses the filter mapper, which first reads the registry and determines the types of filters available. The filter graph manager then attempts to link filters that accept that data type until it reaches a rendering filter. A merit value is registered with each filter and, of the filters that might be capable of handling the data, the filters with the highest merit are tried first.

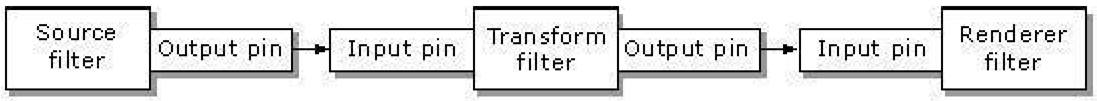
Controlling the media stream means starting, pausing, or stopping the media stream. It can also mean playing for a particular duration or seeking to a particular point in the stream. The filter graph manager allows the application or ActiveX Control to specify these activities, and then calls the appropriate methods on the filters to invoke them. It also allows filters to post events that the application can retrieve. Therefore an application can, for example, retrieve status about some special filter it has installed.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

Filters and Pins

The two basic components used in the stream architecture are filters and pins. A *filter* is a COM object that performs a specific task, such as reading data from a disk. For each stream it handles, it exposes at least one pin. A *pin* is a COM object created by the filter, that represents a point of connection for a unidirectional data stream on the filter, as shown in the following illustration.



Input pins accept data into the filter, and *output pins* provide data to other filters. A source filter provides one output pin for each stream of data in the file. A typical transform filter, such as a compression/decompression (codec) filter, provides one input pin and one output pin, while an audio output filter typically exposes only one input pin. More complex arrangements are also possible.

You can name pins anything you want. If your pin name begins with the tilde (~) character, the filter graph will not automatically render that pin when an application calls IGraphBuilder::RenderFile. This can apply to pins that are just informational and are not meant to be rendered, or need to be enumerated so that their properties can be set. The tilde (~) prefix only affects the behavior of RenderFile and intelligent connect (IGraphBuilder::Connect). Note that intelligent connect can still be used to connect pins with this property if they implement the IPin::Connect method. However, output pins of intermediate filters which are being used to complete the connection which have the tilde at the start of their name will not be connected as part of the intelligent connection attempt.

At a minimum, a filter exposes the IBaseFilter interface. This interface provides methods that allow the enumeration of the pins on the filter and return filter information. It also provides the inherited methods from IMediaFilter; these methods allow control of state processing (for example running, pausing, and stopping) and synchronization, and are called primarily by the filter graph manager.

In addition, a filter might expose several other interfaces, depending on the media types supported and tasks performed. For example, a filter can expose the ISpecifyPropertyPages interface to support a property page.

Pins are responsible for providing interfaces to connect with other pins and for transporting the data. The pin interfaces support the following:

- The transfer of time-stamped data using shared memory or other resource.
- Negotiation of data formats at each pin-to-pin connection.
- Buffer management and buffer allocation negotiation designed to minimize data copying and maximize throughput.

Pin interfaces differ slightly, depending on whether they are output pins or input pins.

An output pin typically exposes the following interfaces.

- IPin methods are called to allow the pin to be queried for pin, connection, and data type information, and to send flush notifications downstream when the filter graph stops.
- IMediaSeeking allows information about the stream's duration, start time, and stop time to be relayed from the renderer. The renderer passes the media position information upstream to the filter (typically the source filter) responsible for queuing the stream to the appropriate position.
- IQualityControl passes quality-control messages upstream from the renderer to the filter that is responsible for increasing or decreasing the media supply.

An input pin typically exposes the following interfaces.

- IPin allows the pin to connect to an output pin and provides information about the pin and its internal connections.
- IMemInputPin allows the pin to propose its own transport memory allocator, to be notified of the allocator that an output pin is supplying, to receive media samples through the established allocator, and to flush the buffer. This interface can create a shared memory allocator object if the connected pin does not supply a transport memory allocator.

The standard transport interface, IMemInputPin, provides data transfer through shared memory buffers, although other transport interfaces can be used. For example, where two components are connected directly in hardware, they can connect to each other by using the

IPin interface, and then seek a private interface that can manage the transfer of data directly between the two components.

Stream Control Architecture

The stream architecture allows applications to communicate with the filter graph manager; it also allows the filter graph manager to communicate with individual filters to control the movement of the data through the filter graph. Using the stream architecture, filters can post events that the application can retrieve, so an application can, for example, retrieve status information about a special filter it has installed.

The filter graph manager exposes media control and media positioning interfaces to the application. The media control interface, IMediaControl, allows the application to issue commands to run, pause, and stop the stream. The positioning interface, IMediaSeeking, lets the application specify which section of the stream to play.

Individual filters expose an IBaseFilter interface so that the filter graph manager can issue the run, pause, and stop control commands. The filter graph manager is responsible for calling these methods in the correct order on all the filters in the filter graph. (The application should not do this directly.)

For position commands, the filter graph manager is called by the application to, for example, play a specified length of media stream starting at some specified stream time. However, unlike the IBaseFilter interface, only the renderer filter exposes an IMediaSeeking interface. Therefore, the filter graph manager calls only the renderer filter with positioning information. The renderer then passes this position control information upstream through **IMediaSeeking** interfaces exposed on the pins, which simply pass it on. The positioning of the media stream is actually handled by the output pin on the filter that is able to seek to a particular position, usually a parser filter such as the AVI splitter.

Position information is passed serially upstream because there might be filters between the renderer and the source filter that require position information. Consider a transform filter that is written to perform some video or audio modification only during the first 10 seconds of a video clip (for example, increasing the volume or fading in the video). This filter probably needs to have information about where the stream is starting so that it can determine its correct behavior. For example, it should not perform if the start time is after the first 10 seconds, or it should adjust accordingly if the start time is within this duration.

Filters also get position information from the IPin::NewSegment method which provides the media start and stop times for the next set of samples and the rate to be associated with those samples.

Quality-Control Management

The Microsoft® DirectShow™ stream architecture provides for graceful adaptation of media rendering to overloaded or underloaded media streams. The IQualityControl interface is used to send quality-control notifications from a renderer filter either upstream, eventually to be acted on by some filter in the graph, or directly to a designated quality control manager. The base classes implement the passing of quality control notifications upstream by providing the **IQualityControl** interface on the output pins of filters. Quality control notification uses a Quality structure, which indicates whether the renderer is overloaded or underloaded. A filter capable of, say, discarding samples to relieve an overloaded condition, can then act on this notification. This is typically done by a source filter but could be done by other filters. For example, the DirectShow AVI Decoder filter skips samples until the next key frame when it receives a quality control notification.

About Capture Filter Graphs

This article provides a brief introduction to capture and introduces the fundamentals of filter graphs that provide video or audio capture or preview capabilities. It includes conceptual diagrams of the most common capture-related filter graphs to help you visualize the components in each filter graph and see how they fit together. It discusses the role of particular filters such as video and audio capture filters, the AVI MUX (multiplexer) filter, and the file writer filter. It also highlights unusual points such as filters (for audio capture in this case) that have input pins.

See Where Can I Learn About... for a list of topics relating to capture, including articles that discuss writing code to perform capture.

If you are new to DirectShow, read Filter Graph Manager and Filter Graphs and Filters and Pins to familiarize yourself with the architecture's fundamental concepts.

Contents of this article:

Capture Introduction

Video and audio capture enable you to take multimedia data from an external source such as a VCR or camera, and view it, listen to it, or save it on your computer's hard drive. Your computer must include video or audio capture hardware to perform capture. For audio capture, a sound card with a microphone or line-in jack is usually sufficient. Some video capture cards support audio capture as well, so you might not need two separate cards. The DirectShow architecture provides default components (filters) that enable you to capture video and audio data streams given the appropriate capture hardware and drivers. DirectShow takes advantage of new capture drivers that are written as DirectShow filters and also uses existing Video for Windows-style drivers.

Video and Audio Capture Filter Graphs

Typical filter graphs that provide video and audio capture and video preview capability must include video capture, audio capture, multiplexer (MUX), file writer, and video renderer filters. If you need a subset of these features your filter graph can be simpler and contain fewer filters. This section begins by discussing the simpler capture filter graphs and the components they contain. It then discusses filter graphs that combine features and build upon the basic capture filter graphs to provide more functionality.

This section contains the following topics:

Video Preview Filter Graphs

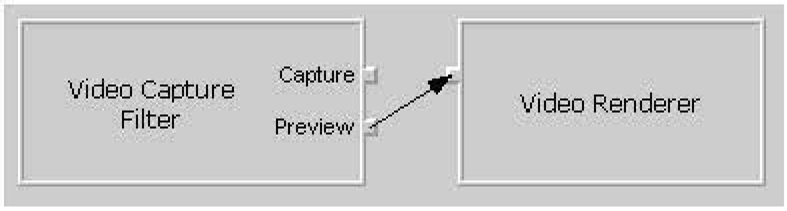
A video preview filter graph enables you to watch the video on your computer screen as it plays from your VCR, camera, or other video source. The video preview filter graph is very simple. It contains a video capture filter and a video renderer. The capture filter provides the video data from the capture card, exposing a pin called Preview to which the video renderer connects. The video renderer provides a playback window in which it displays the video data. If the capture filter produces compressed data, you must insert a decompressor filter between the capture filter and the renderer.

Note Capture filters are not required to expose a preview pin.

The portion of the graph from the preview pin downstream to and including the video renderer is called the *preview section* of the filter graph.

DirectShow provides video capture and video renderer filters. If you have an existing Video for Windows-style capture driver, the video capture filter wraps the functionality provided by that driver so that it works with the DirectShow architecture. You can also write your own video capture filter or use third-party capture filters.

The following diagram illustrates a simple video preview filter graph.



Video capture filters expose a pin for capture as well as an optional pin for preview. Pin names can vary from those shown in the diagram above. The next section of this article discusses the capture pin.

Note Some capture filters have a capture pin and do not have a preview pin. If the capture pin is the only pin on a capture filter, connect the renderer to the capture pin.

Video Capture Filter Graphs

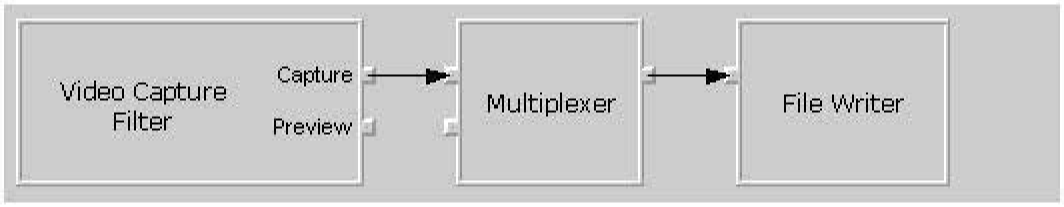
A video capture filter graph takes captured video data and saves it to a file. The term "video capture filter graph" often encompasses video capture and preview functionality, but this section uses the term in the strict "capture to file" sense.

The simplest video capture filter graphs contain a video capture filter, multiplexer filter, and file writer filter. The capture filter provides video data from the capture card, just as it does in video preview filter graphs. It exposes a pin for capture to which the multiplexer filter (MUX) connects. The multiplexer filter understands a particular file format, such as audio-video interleaved (AVI). It has multiple input pins and one output pin. Each input pin takes in a stream of audio or video data. The MUX combines the separate streams of data into the appropriate file format and then passes the newly combined data through its output pin to the file writer. The file writer filter writes the data stream from the MUX to disk without any knowledge of the particular data format.

The MUX and file writer work together as a unit.

DirectShow provides the AVI MUX and File Writer filters. The AVI MUX filter packages data streams into an AVI file stream; therefore, the File Writer always writes files in AVI format if it is connected to the AVI MUX.

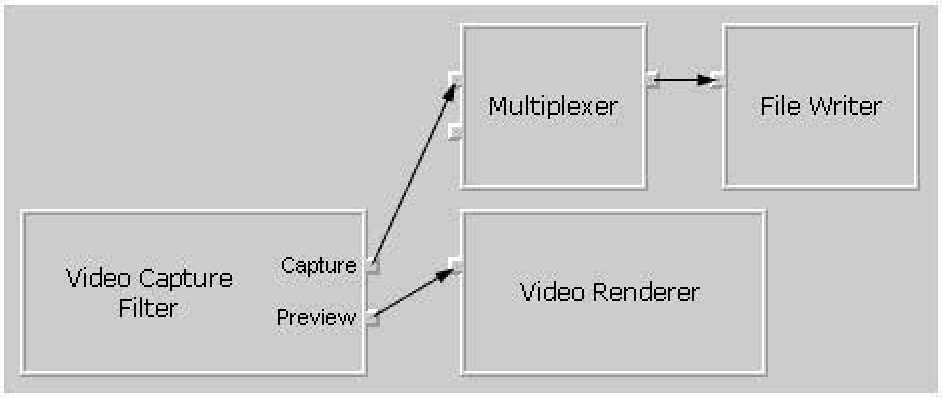
The following diagram illustrates a simple video capture filter graph.



Video Capture and Preview Filter Graphs Combined

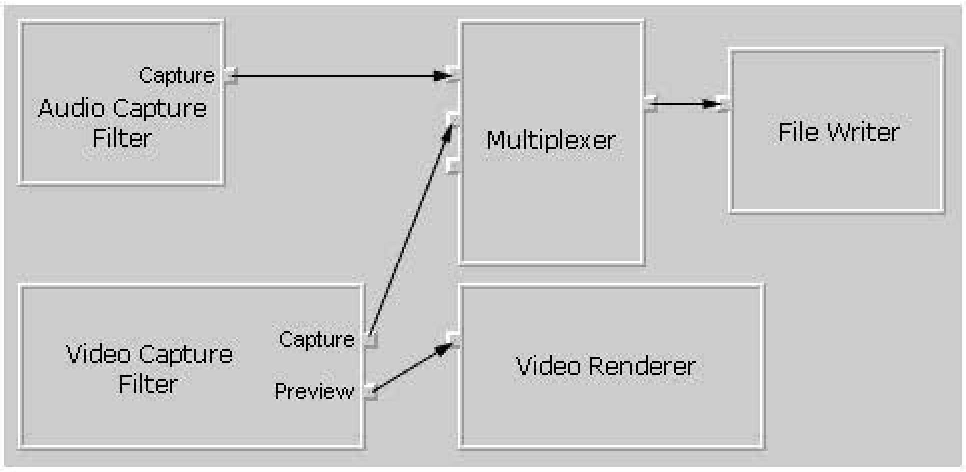
A filter graph that provides both video capture and video preview capabilities combines all the filters from video capture and video preview filter graphs. All the filters serve the same role as in the smaller filter graphs.

The following diagram illustrates the combined video capture and video preview filter graph.



Adding Audio Capture

None of the filter graphs discussed thus far capture audio data. As a result, they produce movies that are silent upon playback. Adding an audio capture filter to the video capture and preview filter graph, as illustrated in the following diagram, provides the missing audio capture capability.



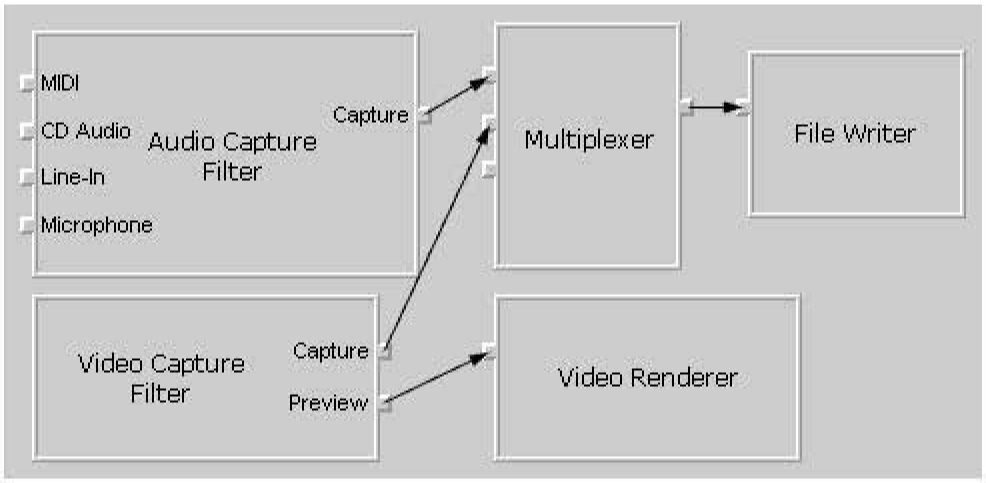
Audio capture filters accept audio data from the audio capture card much as video capture filters accept video data. A capture card might provide both video and audio capture capabilities, so the corresponding video and audio capture filters might process data from the same capture card. If your system contains separate audio and video capture cards, the video capture and audio capture filters process data from separate capture cards.

Audio capture filters also expose a capture pin that connects to the multiplexer filter in the same way that the capture pin on a video capture filter connects to the multiplexer. The multiplexer's role becomes more important in this filter graph because it has more than one connected input pin. Each connected input on the MUX provides a separate data stream (one video and one audio in this case), which the MUX combines into its supported file format, and the file writer saves the resulting data to a file on disk.

Capture Filter Input Pins

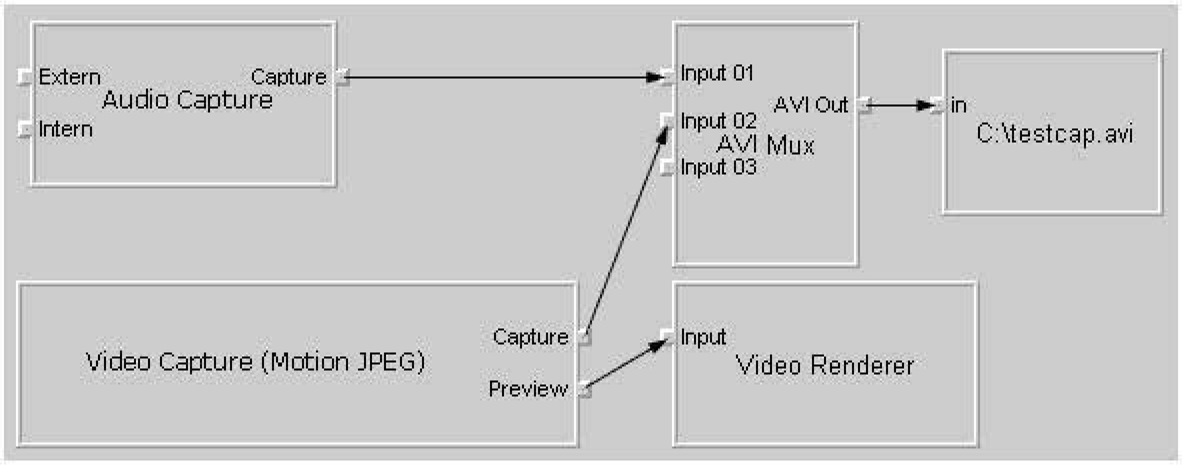
An unusual feature of audio and video capture filters is that they can expose input pins, unlike other source filters. Source filters do not typically expose input pins because they are the source of the data. They typically pass data on to the next filter in the graph rather than accepting input data from another filter. The input pins provide a mechanism to access input characteristics. Audio capture filters support the IAMAudioInputMixer interface to provide access to such characteristics as recording and bass levels on each input line. Each input pin represents an input line such as microphone, CD audio, or MIDI on the audio card.

The following diagram shows a full-featured filter graph that provides video preview, and video and audio capture. The audio capture filter exposes an input pin for each input line on the capture card. Internally, each pin supports IAMAudioInputMixer to enable applications to access input characteristics on each line.



Example Capture Filter Graph

Now that you're familiar with capture filter graphs in general, here's a screen shot of a capture filter graph from the Filter Graph Editor tool that is included with the DirectShow SDK. It builds upon the conceptual diagrams examined earlier.



The preceding screen shot shows a motion JPEG video capture filter and an audio capture filter in a capture filter graph. Both filters process data from the same capture card because the capture card happens to capture both video and audio data streams. A computer might include a sound card in addition to a capture card, giving you a choice between two audio sources and therefore between audio capture filters to insert in the capture filter graph.

The screen shot is very similar to the conceptual diagrams examined earlier with the exception of filter names and input pin names. The input pins on the audio capture filter are labeled Intern and Extern. The file writer filter saves captured video and audio data to a file called Testcap.avi at the root of the C: drive.

If you have capture hardware installed on your system and use the Filter Graph Editor tool to create a capture filter graph, it will appear similar to the preceding screen shot. The names of the video and audio capture filters depend on the capture drivers installed on the computer. If present, the audio capture filter's input pins might have different names from those illustrated. The name of the file to which captured data is saved will differ as well. The preview pin will connect to a decoder if the data from the pin is compressed, and the decoder will then connect to the renderer.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

Improving Capture Performance

Capturing is a hardware-intensive operation that requires saving a large amount of information to disk as quickly as possible. This information is typically in the form of video and audio data. Reducing bottlenecks that slow down the system is very important, because it can help improve the quality of the captured movie.

This article presents some general suggested practices and hints and tips that can help you, as the user of a capture application, set up the capture system for optimal capture performance.

Note Be sure to read your capture card manual for information specific to your capture card. Systems vary as well, so all the information presented in this article might not apply to your system configuration.

See Capture Introduction for a brief introduction to capture. See capture topics for a list of Microsoft® DirectShow™ interfaces and articles relating to capture.

Contents of this article:

Capture Numbers

Capturing involves transferring a large amount of data from a capture card to disk. To get an idea of the amount of data and the data throughput required for a particular capture scenario, consider the following:

Suppose you want to capture a movie with a height of 320 pixels, a width of 240 pixels, a capture rate of 30 frames per second (fps), and in 24-bit color format. The movie does not include any sound.

Use the following formula to determine the number of bytes of uncompressed data that must be transferred every second in order to capture all of the image data, and therefore to maintain the image quality.

```
bytes of transferred video data = height (in pixels) x width (in pixels) x rate
(in fps) x color depth (in bytes)
```

Plugging in the numbers from the preceding scenario produces the following result.

```
320 x 240 x 30 x 3 = 6912000 bytes of transferred video data
```

Capturing one second of this movie at the desired size, frame rate, and color depth requires approximately 6.9 million bytes of disk space. Multiplying by 60 seconds produces the results for a minute; each minute of capture requires 414,720,000 bytes in this case. You can reduce the amount of data required by reducing any of the parameters in the above formula: capture a smaller image, at fewer frames per second, or with fewer colors. However, in cases where your image requirements push the system to its limits and you need to be able to capture at the highest possible number of megabytes per second, you'll want to optimize your system as much as possible. After all, the capture settings you use affect how the movie will look when someone plays it back.

The preceding numbers are for a silent movie. If you want to capture audio as well as video, you have to add the amount of required audio data. For example, CD-quality audio, recorded at 44 kilohertz (kHz), 16-bit, stereo, requires about 172 kilobytes (KB) per second. Audio capture is also very CPU-intensive, and synchronizing the audio and video data (to achieve proper lip synch, for example) can cause delays as well.

You might find that your system can't keep up with the required amount of data transfer for the settings you've chosen. When capturing, your system might pause, the video might be jerky or jitter (not smooth), and some of the frames might be dropped (not saved to disk). Playback quality of such an image is typically unsatisfactory. To avoid such problems, you can follow a number of practices to optimize your system for capturing.

The suggestions presented in this article can help you reach the goal of optimal capture performance. At the same time, bear in mind that each system is different and something that improves performance on one system might not be effective on another system.

Optimizing the Hard Disk for Capture

Because capturing is very hard-disk intensive, optimizing the files on the hard drive that you'll use for capturing (also called the *data drive* or *data disk*) is the most important task in optimizing capture performance. The following list contains goals in optimizing the data drive and techniques you might use to achieve those goals. The techniques discussed here are suggestions and might not be helpful given particular capture requirements. Your requirements and resources govern precisely which techniques you might want to try.

- Ensure the capture file is in a contiguous (nonfragmented) location on the data drive.

  The heads of a hard drive can read from and write to a contiguous file more efficiently than if they have to seek to other, nonadjacent portions of the disk. Use a tool such as the Microsoft Windows® Disk Defragmenter (Defrag.exe) to defragment your data disk. Defragment both the data drive and the operating system drive. The operating system drive comes into play when using drivers (such as audio and video drivers), writing to the system cache, writing to the registry, using overlays, and so forth. Run the Windows 95 Scandisk tool to ensure the integrity of the data drive and the operating system drive.

- Preallocate a capture file that is larger than any movie clips you expect to save.

  Allocating file space is time-consuming, so you should allocate your file before you capture. Capture software such as the DirectShow AMCap sample lets you allocate space for the capture file. If you capture more data than will fit in the capture file, the system has to allocate more space for the file as you capture, which, again, slows down capture. Avoid the reallocation of file space during capture, and the speed penalty, by allocating a file that is large enough to meet your needs. Saving the captured data can require as much space as the original capture file, so ensure you have enough free hard disk space to save your captured data to another file. Be sure to regularly defragment your capture file as well.

- Devote an entire hard disk, or partition on the disk, for the capture file.

  This technique is particularly useful under Windows NT® because Windows NT does not include a Scandisk or Disk Defragmenter tool. Reserving an entire disk or partition on the disk for the capture file can make it easier to keep the capture file space clean and contiguous. You can format such a disk or partition, and then preallocate file space again, or defragment it without having to worry about other files on the disk or partition. When you format a dedicated capture drive, use the full format to initialize the disk rather than a quick format that leaves old data on the disk.

  Save your captured images to a directory that is not on your data drive or data partition to help keep your data drive clean. If you can't devote an entire drive to capture, allocate space for the capture file, defragment the file, and (in Windows 95) run the Windows 95 Scandisk tool.

- Place the capture file at the beginning (outer rim) of the data drive.

  If you allocate the capture file as the only file on the disk, or as the only file in the first partition on the disk, it will begin at the outer rim of the disk. Access to the outer portion of a hard disk is faster than access to the inner portion of the disk. If you don't have a hard drive to devote to capture, you can use disk utilities to move your capture file to the beginning of the disk.

Revisit these goals as necessary before each capture session to ensure your disk is configured for optimal performance. Defragment the data disk before each capture session and defragment the drive containing your saved images before you play back the saved files.

Disk Settings in Window 95 and Later

The System applet of the Windows 95 and OSR2 Control Panel contains several options you can disable for optimal capture performance. In high-bandwidth situations like capture, it's important to make sure the drive is writing as much data as possible and not spending time with software optimizations or checking for system changes. The options to disable include the following:

- Automatic detection that a CD-ROM disc has been inserted in the CD-ROM drive
- Read-ahead optimization for the hard disk
- Write-behind caching for all drives

To access these options in Windows 95, click the Start menu. Under Settings, click Control Panel, and double-click the System applet. Select the appropriate tab and proceed as outlined below:

- Device Manager tab: Click **CD-ROM** and click your CD-ROM drive. Click the **Properties** button, select the Settings tab and clear the **Auto insert notification** check box.
- Performance tab: Click the **File System** button and drag the **Read-ahead optimization** slider to None. (The default is Full.) While still on the File System Properties dialog, select the Troubleshooting tab and check **Disable write-behind caching** for all drives.

You will have to restart your machine for the new settings to take effect.

Note For optimal performance for other applications, be sure to return these settings to their original values after your capture session is complete.

Reducing Noncapture-Related Machine Activity

Anything that interrupts the system or consumes CPU time for purposes other than capture can potentially decrease capture performance. Consider performing the following tasks to see if they affect performance on your system.

- Close all applications except the capture application.
- Turn off the clock that Windows 95 displays on the taskbar. To do so, right-click the taskbar, click **Properties** and clear the **Show Clock** check box on the Taskbar Options tab.
- Turn off the screen saver. To do so, right-click the desktop, click **Properties**, select the Screen Saver tab, and choose "(None)" from the screen saver drop-down combo box.
- Turn off your printer.
- Disable your network card if you have one. Sending and receiving data over the network can interrupt the system, even if you aren't actively doing anything over the network.

Additional Hints and Tips for Improving Capture Performance

This section contains a collection of hints and tips for improving capture performance that you might want to try after experimenting with the other suggestions in this article. The suggestions are grouped according to hardware and software-related suggestions.

Software

- Consider capturing on Windows NT, because the Windows NT file system (NTFS) is typically faster than the traditional FAT file system, due to its use of threads. You might need to contact the manufacturer of your capture card for a driver that will work on Windows NT. Use a dedicated NTFS drive rather than a drive that is part NTFS and part

FAT. The FAT-32 file system is typically faster than the FAT file system.
- Using your capture software, experiment with different compression ratios (for example 2:1 or 1:1) to decrease the amount of data that has to be saved. Start with the default compression ratio and increase it until you drop frames. Try three passes and take the best results of the three. After you've saved the captured image to another drive, run Scandisk on the standard setting to quickly defragment the drive.
- If you installed Windows 95 over your Windows 3.1 installation, put the line "verify=off" in your Autoexec.bat file. This line will prevent DOS from re-reading data after a write operation to make sure the correct data was written. Verification slows down the writing operation.

Hardware

- Insert the capture card in PCI slot zero so it will be checked for activity before other cards on the system.
- Make sure the hard drive cache is turned on for the data drive. Refer to your SCSI card manual for more information.
- Heat buildup inside a system can wear down the system components and decrease capture performance. If you will be capturing continuously for hours at a time, make sure your system has three fans: one each for the power supply, CPU, and components (cards). If your captured images look fine at first, but become jittery after the system has been capturing for a while, the capture card might be overheating.
- Some capture cards include a built-in audio card. Whether you use the on-board audio card or a separate audio card depends on your needs. You might find that a separate audio card provides features you need, or the on-board audio card might suit you just as well.
- Some capture cards have an external connection for a monitor so you don't have to go through software to preview what you're capturing. That feature can help with performance, because the system isn't busy with the preview window.

Choosing a Capture System

Capturing is possible with a wide range of systems and capture cards. Shop around to compare capture cards and features to see what best meets your needs. See Suggested Capture Reading for possible sources of information. The optimal hardware configuration varies depending on the capture card.

If you're setting up a new machine to devote to capture, consider a 166-megahertz Pentium or later, with 64 megabytes or more of EDO RAM (as much RAM as possible), and a 2-gigabyte or larger Wide SCSI 2.0 AV-certified hard disk. AV-certified drives are designed for high-bandwidth data transfer. The SCSI hard disk controller should support PCI bus mastering 2.0 and later, which uses 32-bit drivers. If your capture card supports overlays, ensure that your video card also supports overlays. Make sure the capture card has drivers for the operating system you plan to use.

Suggested Capture Reading

This section lists possible sources of information about capture.

- http://www.ccs.queensu.ca/pubs/itsnote/VideoCapture.html contains a general introduction to video capturing.
- http://gcunix.gc.maricopa.edu/~IC/vidph/vidph05.html contains information about organizing the capture process.

- ☐ http://fre.www.ecn.purdue.edu/FrE/asee/fie95/3a2/3a25/3a25.htm contains an article titled "Effective Video Capture Techniques for Educational Multimedia."
- ☐ http://www.worldguide.com/Tech/videocapture.html contains information about setting up a system for video capture and compression.
- ☐ http://cctpwww.cityu.edu.hk/public/graphics/g3_vidcap.htm contains information about some capture card and chip manufacturers.
- Search the World Wide Web for "capturing".
- Contact the manufacturers of various capture cards, many of whom are available on the Web.
- See the manual for each capture card for its particular requirements.
- For general background regarding digital video, see "PC Video Madness!", by Ron Wodaski, Sams Publishing, Carmel, Indiana, c. 1993, ISBN 0-672-30322-1.

Data Flow in the Filter Graph

This article examines the types of data, including samples, events, and notifications, that move through a filter graph, including where this data and information originates, where it is routed, and the protocols that must be followed for data to flow correctly.

Contents of this article:

How Data Moves in a Filter Graph

Data flow in the filter graph can be viewed by examining the paths through which it flows, and also by examining the protocols that are used within those paths. Data flows primarily in the following paths.

- Media sample data flows from one filter to the next — originating at a source filter and terminating, eventually, at a renderer filter.
- Control information, such as end-of-stream and flushing notifications, travels with the

media data stream from filter to filter.
- Event notification events flow from the filters to the filter graph manager and, optionally, to the application.
- Filter graph control data flows from the application to the filter graph manager and finally to the filters themselves.
- Quality control data originates in the renderer and flows upstream through the filters until it finds a filter capable of increasing or decreasing the media data flow. It might also flow directly to a quality control manager if one is registered.

This article describes each of these data paths. Data movement in a filter graph is enabled by implementation of the following Microsoft® DirectShow™ filter graph protocols.

- *Media sample protocol*, which defines the way that media samples are allocated and passed between filters.
- *End-of-stream protocol*, which defines how filters generate and process end-of-stream information and how the filter graph manager is notified.
- *Flushing protocol*, which defines how filters flush data through the filter graph.
- *Error detection and reporting protocol*, which defines how errors are handled by filters and propagated to the filter graph manager.
- *New segment protocol*, which defines a means to enable start and stop times and data rate information to be presented to a filter in advance of the data, so that a filter can adjust its processing accordingly.
- *Quality management protocol*, which defines how the filter graph adapts dynamically to hardware and network conditions to degrade or improve performance gracefully.

Media Sample Data Flow

DirectShow filters pass media data downstream, that is, from the output pin of one filter to the input pin of the next filter. The flow and control of the data is effected by the interfaces on those pins and the filters themselves. The filters serialize data streaming activity; all data streaming calls for a given pin are explicitly serialized and usually originate from a single thread.

Data starts at a source filter and ends at a renderer filter. The source filter can either push the data down the graph (that is, originate the thread and send data to the IMemInputPin::Receive method of the downstream pin), or implement the IAsyncReader interface and let the downstream filter originate the thread, pull the data from the source filter, and send it downstream. For a description of how this latter case differs from other protocols, see IAsyncReader Transport.

Every filter should accept and process data received by its input pins, with the following exceptions.

- The filter is in a stopped state.
- The pin is in the middle of a flush operation. That is, the pin's IPin::BeginFlush method has been called but its IPin::EndFlush method has not been called yet (see Flushing).
- The input pin rejected some previous data and no flush or stop action has occurred since (in which case the connected output pin should have stopped sending data anyway).

There can be other conditions for a filter to reject data as well. For instance, a transform filter would reject data at its input pin if its output pin was not connected.

Media samples are data objects that support the IMediaSample interface. They are usually obtained from an allocator, which is most likely represented by an object supporting the IMemAllocator interface. The two connected pins of adjacent filters agree on a common method of exchanging data, called a transport. Many of the base classes for the DirectShow class library are used to implement objects that support the local memory transport.

In the local memory transport, the input pin for a connection supports the IMemInputPin interface. An output pin can determine that it can use the local memory transport if a call to the IUnknown::QueryInterface method on the input pin to request the **IMemInputPin** interface succeeds. For this transport, data is passed from the output pin of one filter to the input pin of an adjacent filter in media samples. During connection, the output pin and input pin agree on the connection's allocator object, which is used to create the media samples.

Filters must follow protocols to pass and receive media samples. The connected pins must agree on the allocator to be used, must have a means of passing the data, and must follow the correct procedure for holding on to a sample or releasing it back to the sender.

For the local memory transport, an output pin passes a media sample to the input pin it is connected to by calling the input pin's IMemInputPin::Receive or IMemInputPin::ReceiveMultiple methods, depending on whether it is passing single or multiple samples. Before it can pass this data, however, the output pin must obtain a media sample. The IMemInputPin interface on the input pin provides an IMemAllocator object when requested to provide an allocator. If the output pin is not using its own allocator, or one provided to it from further upstream, it calls the IMemAllocator::GetBuffer method on the input pin to retrieve one.

The input pin can either process the data right away or save it for later processing. In the latter case, it must call the IUnknown::AddRef method on the media sample to prevent the sample from being returned to the allocator. When the output pin has called the input pin's IMemInputPin::Receive method, it must call the IUnknown::Release method to free the sample. If the input pin did not save the sample by calling **IUnknown::AddRef**, the sample is immediately returned to the allocator.

The output pin can decide not to pass the media sample on to the input, in which case it can just call the sample's IUnknown::Release method without calling the input pin's IMemInputPin::Receive method.

Control Information Data

There are two types of control information which are passed downstream filter to filter:

- End-of-stream notifications
- Flushing

End-Of-Stream Notifications

It is important for filters to indicate when there will be no more data in the current set of data. A filter does this by sending an end-of-stream notification to the next filter, which is accomplished by the output pin calling the IPin::EndOfStream method on the downstream filter's input pin.

When a source filter (an originator of data) reaches the end of its data, it calls the IPin::EndOfStream method on all pins connected to its output pins. This mechanism is propagated down the filter graph so that each filter that processes its EndOfStream method in turn calls **EndOfStream** on the pins connected to its output pins. When the notification reaches the end of the line in the filter graph, the renderer passes an EC_COMPLETE notification to the filter graph manager. The filter graph manager counts the **EC_COMPLETE** notifications it receives and when all renderer filters have completed, passes the notification to the application. The filter graph manager counts rendered streams by counting the number of filters (not pins) that support IMediaSeeking or IMediaPosition and have a rendered input pin. A rendered input pin is a pin with no corresponding outputs, which can be determined with IPin::QueryInternalConnections. input pins. A renderer input pin returns zero pins when its **IPin::QueryInternalConnections** is called. Note that the filter, not the pins, support **IMediaSeeking** in this case.

Although source filters usually originate the end-of-stream notification, it is also possible for other filters to detect this condition and generate the notification downstream. Most notably, this applies to parser filters that connect to asynchronous reader filters (filters implementing the IAsyncReader interface).

For example, the MPEG parser (in the MPEG splitter filter) can detect the end of the stream and when it does, return S_FALSE from the Receive method, which signals the upstream filter to stop sending data until a seek occurs or the filter graph is stopped. In this case, the upstream filter does not need to call EndOfStream. Instead, **EndOfStream** should be called by the filter detecting the end-of-stream condition (the downstream splitter or parser) before returning from **Receive** or ReceiveMultiple.

Note that EndOfStream should be serialized with data passed in the stream. It is a single piece of information that must be passed after all the other data in the stream.

Flushing

In the DirectShow filter graph architecture, flushing is a two-stage process. Flushing is not usually initiated as part of normal data flow, but rather as a result of a control action from the filter graph manager. As such, it is an asynchronous event which requires flushing of old data followed by a resynchronization and pushing of new data.

In a flushing operation, first IPin::BeginFlush is called by the source filter on all input pins connected to its output pins. This call is propagated down the graph by all filters to the rendering filter or filters. BeginFlush should flush any pending EndOfStream calls or EC_COMPLETE notifications. After **BeginFlush** has been called, an input pin should reject all data until its IPin::EndFlush method has been called (this includes end-of-stream notifications). It should also free any connected pins waiting for any of its resources. In the case of the local memory transport, this means that every filter should free any filter waiting for a sample from its allocator. This is usually done by calling IMemAllocator::Decommit on the allocator.

After a filter has called BeginFlush on the pins connected to its output pins, and when it can guarantee that all processing of samples by its pins is complete and no more samples will be processed, it should call EndFlush. For source filters this means shutting down data generation, then calling **EndFlush** on the pins connected to its output pins. For other filters it means waiting for an **EndFlush** call (which guarantees that no more samples will be sent) and then waiting for any queues it maintains itself to empty. Because calls can block on downstream filters for the local memory transport model, it is important to wait for queues to empty when **EndFlush** is called, rather than trying to do so when **BeginFlush** is called.

Event Notifications

Notification data goes from filters to the filter graph manager and can be passed on to the application. The EC_COMPLETE notification, which is sent from renderers at the end of a data stream, has already been mentioned.

The filter graph manager should not be notified of EC_COMPLETE until a Run command is issued. A renderer filter that has EndOfStream called on its input pin while in a paused state must not notify the filter graph manager until its IMediaFilter::Run method is called. Stop and EndFlush calls cancel any such deferred notifications and allow more data to be subsequently processed by the pin. After notifying the filter graph manager once with **EC_COMPLETE**, the renderer must not generate another **EC_COMPLETE** notification before processing a **Stop** or **EndFlush** method.

If a running filter graph is paused while at the end of its stream and IMediaFilter::Run is subsequently called, renderers should notify the filter graph manager with EC_COMPLETE again.

Besides EC_COMPLETE, there are many other event notifications, many of which are sent by specialized filters, such as the renderer, to communicate with a host application. Error notifications are another class of notifications that are also sent from filters to the filter graph manager.

The convention for DirectShow filters is that when a filter detects an error, it passes a notification to the filter graph manager by calling the IMediaEventSink::Notify method. Errors in processing data can generate several error events, including the following:

- EC_STREAM_ERROR_STOPPED, if no more data can be processed.
- EC_STREAM_ERROR_STILLPLAYING, if data can still be processed.

If processing can no longer continue, the filter graph manager should be notified with EC_STREAM_ERROR_STOPPED and the appropriate convention for the particular transport should be used to notify the upstream output pin. In the case of the local memory transport, this involves returning an error value from IMemInputPin::Receive. In addition to notifying the filter graph manager of the error, a filter should also either call EndOfStream on all the pins connected to its output pins or, if it is a renderer, also notify the filter graph manager with EC_COMPLETE. This ensures the play will complete gracefully.

Errors of this type can be caused by encountering events such as being out of memory or other resource problems. Or they might be caused by other events such as a failure to obtain a buffer when trying to pass data downstream.

On the other hand, when an error occurs but processing can still continue, EC_STREAM_ERROR_STILLPLAYING should be sent to the filter graph manager. In this case, the upstream output pin should not be notified. Specifically, for the local memory transport, the input pin's IMemInputPin::Receive method should return NOERROR when this type of error occurs.

Filter Graph Control Data

Control data originates at the application and is passed to the filter graph manager. At the COM level, this is handled by filter graph manager interfaces in the Control.odl type definition library. Examples of control data are calls to the IMediaControl interfaces, such as IMediaControl::Run, IMediaControl::Pause, and IMediaControl::Stop. The IMediaPosition and

IMediaSeeking interfaces provide a means of moving forward or backward in a media stream.

The most important thing to understand about the flow of control data is that it should always pass through the filter graph manager first. This is because there is usually an order that must be followed in controlling the filters in the filter graph to make sure filters are called in the correct order and with regard for internal filter graph states.

Quality Control Data

The DirectShow stream architecture provides a means of gracefully adapting to load differences in the media stream so that rendering of the data is maintained at the highest possible resolution. The IQualityControl interface is used to send quality-control notification messages from a renderer filter either upstream, to be acted on eventually by some filter in the graph, or directly to a designated location. For example, a renderer that is getting too many frames to process can try to get an upstream filter to cut down on the number of frames it is sending. This is usually more efficient than simply dropping frames at the renderer. (A video decompressor filter uses many CPU cycles to decompress frames, so it is better to discard samples before processing them rather than after processing them.) Likewise, when the renderer filter can handle more data, it sends notifications to increase the number of samples.

Quality-control messages are passed upstream by default; if a filter has no registered quality sink, the default action passes the message to the IQualityControl interface of the connected output pin. Internally, the output pin passes the quality-control message to its input pin, if it has one, and the message travels upstream until it reaches a filter that can affect the data stream quality in the requested manner. DirectShow handles this mechanism automatically in the transform base classes.

If a filter can handle the quality notification (by increasing or decreasing the flow) and if it is not appropriate for filters further upstream to take any action, that filter will act on the notification and not pass it on. A filter must pass the quality-control message on if it does not act on the message. It can also pass it on even if it does act on the message. Silently accepting the message without acting on it or passing it on is considered bad behavior, and might damage the performance of the filter graph as a whole.

A *quality sink* is a feature implemented by the IQualityControl::SetSink method. When this method is called, the filter is instructed not to send messages upstream, but rather to send them to the object passed to the SetSink method. Typically, this object would be a component called a quality-control manager. Such a component would set itself as the sink for all the filters to send the quality-control messages rather than anywhere else. It would then determine whether to route the messages upstream or to take some other action, such as cutting back the video stream to avoid breaking the audio. A quality-control manager can be implemented by using the IQualityControl interface and should be anticipated when writing filters.

Serializing Data

A filter usually has to synchronize two contexts: filter state and data flow. Usually a filter will create a critical section for each context.

The data flow critical section is held during streaming operations. For example, for the local memory transport, this critical section should be held while processing the following methods.

- IPin::NewSegment

- IMemInputPin::Receive
- IMemInputPin::ReceiveMultiple
- IPin::EndOfStream

The filter state critical section is held while processing state changes when the following IBaseFilter methods are called.

- Stop
- Pause
- Run

It is also held during BeginFlush and EndFlush streaming control operations.

During Stop and EndFlush calls, the stream state must be synchronized with the filter state. An example of how to do this is in the base class CTransformFilter. In the case of implementing the **Stop** method for the local memory transport, for example, the stream must be "released" to avoid deadlocks by decommitting the input allocator pin. This is not required to process **EndFlush**, because this will have already been done in BeginFlush processing. Once the stream is released, the data flow critical section (as implemented in Receive) can also be locked to synchronize with the stream state.

Note that because Stop requires access to the filter state before synchronizing with the data flow component, these two critical sections must be different.

A filter should not, in general, have its filter state critical section locked while calling methods on other filters. The filter graph synchronizes graph-wide operations such as setting the current position.

IAsyncReader Transport

For source parsing filters, the IAsyncReader interface helps implement a "pull" data flow model, as opposed to the "push" model, in which a thread in the source filter pushes data downstream. The parsing filter is connected downstream to the filter whose pin implements **IAsyncReader**. In this case, the downstream parsing filter initiates data transfer by calling **IAsyncReader** methods such as SyncReadAligned. The parsing filter, in this case, creates the thread, pulls data from the source, and pushes it downstream.

Because all data flow activity in this transport is initiated by the downstream filter, several of the protocols mentioned previously operate in reverse. For example:

- The downstream pin initiates BeginFlush and EndFlush upstream during seek operations.
- The downstream pin reports errors to the filter graph manager.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

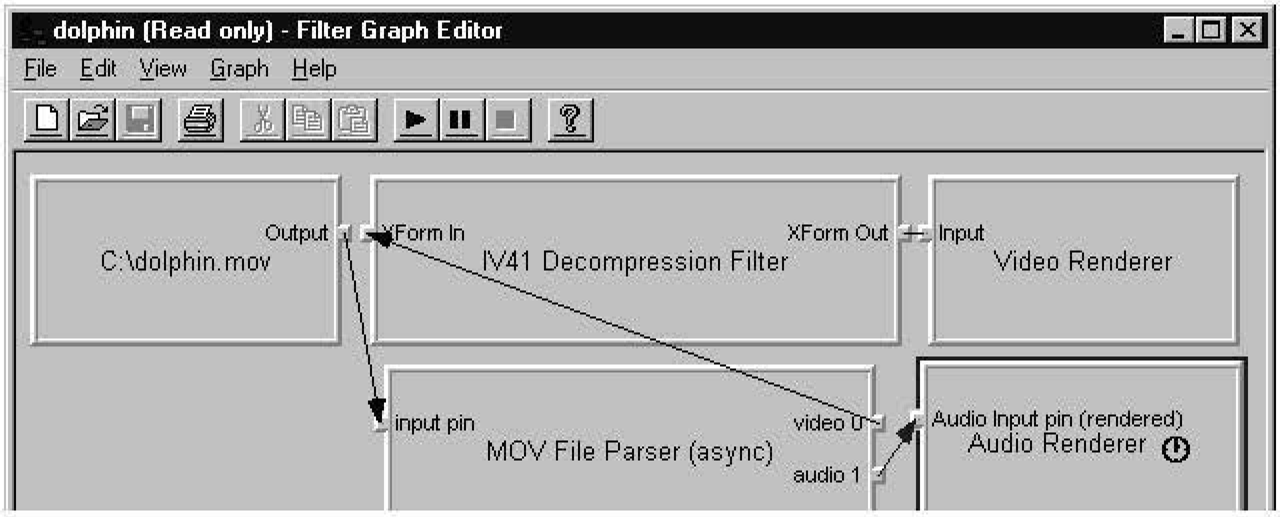
Constructing Filter Graphs Using Visual Basic

This article describes how to use ActiveMovie Control type library objects to manage the components of a filter graph — filters and pins — in applications based on Microsoft® Visual Basic®, version 5.*x*. It also describes the functionality demonstrated by the Builder sample application. This article is written for the Visual Basic developer who is already familiar with application programming in Microsoft Windows®, Windows-based multimedia programming, and Automation features of the Visual Basic programming system.

Contents of this article:

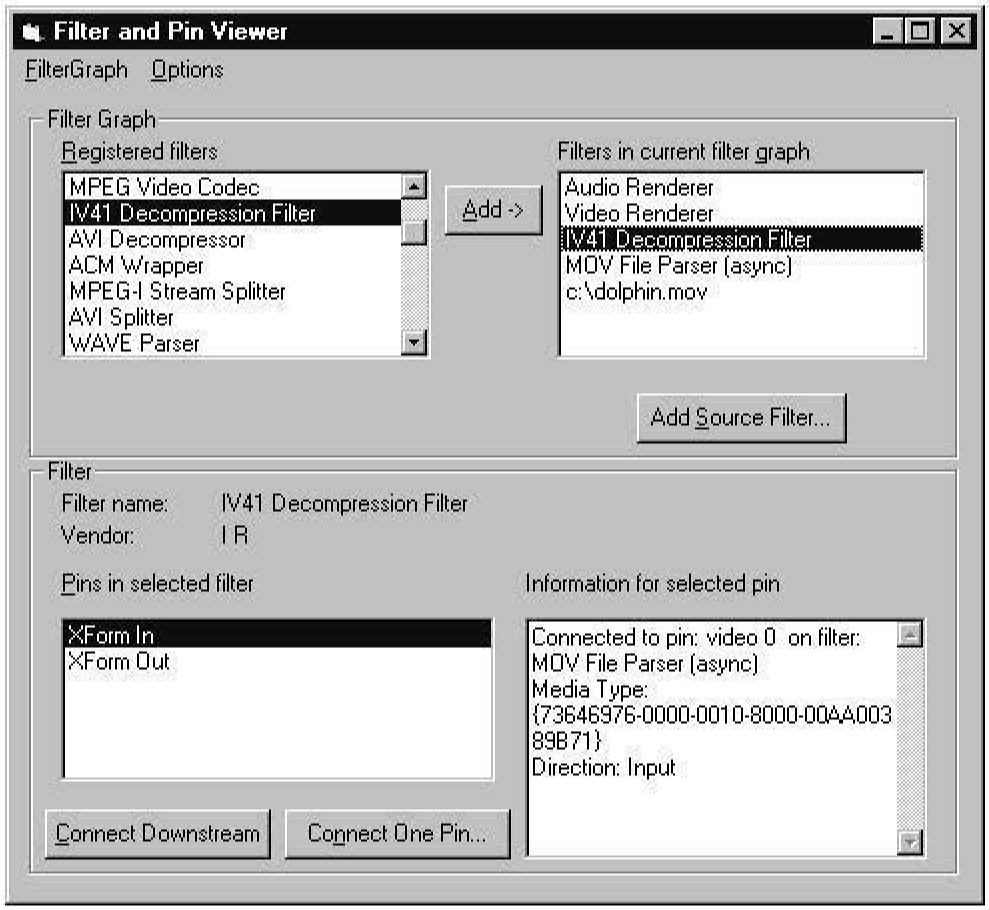
About Filter Graphs

The *Filter Graph Editor* provided with the Microsoft DirectShow™ Software Development Kit (SDK) is a graphical tool that creates and manages filter graphs. There are three types of filters: source filters, transform filters, and renderer filters. The Filter Graph Editor is a graphic user interface you use to construct filter graphs by inserting filters and creating connections between them. For example, you can render the filter graph for a specific multimedia source file, such as Dolphin.mov, and see each of the connections in the graph, as in the following example.


dolphin (Read only) - Filter Graph Editor
File Edit View Graph Help
C:\dolphin.mov
Output
XForm In
IV41 Decompression Filter
XForm Out
Input
Video Renderer
input pin
MOV File Parser (async)
video 0
audio 1
Audio Input pin (rendered)
Audio Renderer

The same filter graph objects, filters and pins, can be managed from a sample Visual Basic-based application that uses Quartz.dll. For simplicity, the Visual Basic-based application uses list boxes rather than graphical elements to show the parts of the filter graph, and displays information for only one filter and only one pin at a time. (The purpose of the application is not to compete with the Filter Graph Editor, but to demonstrate how to retrieve and manage the same underlying filter graph information using Visual Basic.)

The following illustration shows the same multimedia source file, Dolphin.mov, as depicted by the Visual Basic-based sample application.



Registered filters	Filters in current filter graph
MPEG Video Codec	Audio Renderer
IV41 Decompression Filter	Video Renderer
AVI Decompressor	IV41 Decompression Filter
ACM Wrapper	MOV File Parser (async)
MPEG-I Stream Splitter	c:\dolphin.mov
AVI Splitter	
WAVE Parser	

Filter name: IV41 Decompression Filter
Vendor: I R

Pins in selected filter:
XForm In
XForm Out

```
Connected to pin: video 0  on filter:
MOV File Parser (async)
Media Type:
{73646976-0000-0010-8000-00AA003
89B71}
Direction: Input
```

The list box labeled "Filters in current filter graph" lists the four filters required for this source file. The list box labeled "Pins in selected filter" lists the two pins defined for the selected Indeo Video R4.1 Decompression Filter: the "XForm In" and "XForm Out" pins. The XForm In pin is selected, so the application displays detailed information about this pin, including its direction and connection information.

The remainder of this article discusses the Visual Basic code you can use to retrieve and manage such filter graph information. In addition to the general-purpose code that works with any source input, the application includes a routine that illustrates a more likely use of these properties and methods—building a specific hard-coded filter graph for your Visual Basic-based application.

This article assumes you have already set up your Visual Basic environment to use Quartz.dll. For more information about setting up your Visual Basic environment, see Controlling Filter Graphs Using Visual Basic.

About the DirectShow Quartz.dll Objects

Quartz.dll provides objects that you can use in your Visual Basic-based applications to manage filters and pins. There is an implicit hierarchy among these objects; that is, you must often access the properties of one object to obtain another object. In the following example, an object that appears indented below another indicates that you obtain that lower-level object from a property or method of the higher-level object.

```
Filter graph object (IMediaControl)
  Filter collection (RegFilterCollection, FilterCollection properties)
    Filter Info object (IFilterInfo or IRegFilterInfo in filter collection)
      Pin collection (Pins property)
        Pin Info object (IPinInfo item in pins collection)
```

The top-level object in the hierarchy is the filter graph object, or the IMediaControl object, which represents the entire filter graph. You can access two properties of the **IMediaControl** object to obtain collections of filter objects. The RegFilterCollection property represents the filters registered with the system. The FilterCollection property represents the filters that are part of the filter graph.

As with other collections accessible to Visual Basic, you can access individual items in the collections by using the Visual Basic **for each...next** statement. The number of items in the collection is indicated by the Count property of the collection.

The filter collection contains IFilterInfo objects. Each **IFilterInfo** object has a Pins property that represents a collection of pins defined for that filter.

The pins collection contains IPinInfo objects. Each **IPinInfo** object includes detailed information about that pin, including its media type and its connection to another pin.

To examine a specific pin on a filter in the filter graph, use the following procedure.

1. Obtain the filter graph object.
2. Use the IMediaControl.FilterCollection property of the filter graph object to obtain the collection of filters present in the filter graph.
3. Search through the filter collection for the specific filter.
4. Use the IFilterInfo.Pins property of the filter object to obtain the collection of pins defined for the filter.
5. Search through the pins collection for the specific pin.
6. Examine the properties of the pin object to find connection information and other information for the pin.

Creating a Filter Graph

There are three distinct ways to use Quartz.dll to create a filter graph; each way offers a different amount of control over the filter graph. These range from automatically generating the entire filter graph to specifying every filter and pin connection. The three approaches are as follows:

- Automatic.

  Generate the complete filter graph from either a multimedia source or a stored filter graph file.

- Semi-automatic.

  Create a new (empty) filter graph, add one or more filters, then automatically generate all filters and connections needed to render a specific pin.

- Manual.

  Create a new (empty) filter graph, add individual filters to the graph, and explicitly add connections between pins.

The sample application, Builder, demonstrates all three approaches. The application's "Generate from input file" command on the FilterGraph menu supports the first approach. The New command (empty) on the FilterGraph menu supports the other two approaches.

Generating the Complete Filter Graph

The following code fragment demonstrates how to generate the complete filter graph based on the multimedia source or stored filter graph. After creating an IMediaControl object that is initially "empty," the application calls the IMediaControl.RenderFile method to build up the complete graph:

```
' fragment from the Filter Graph menu's Generate from input file command
    ' start by creating a new, empty filter graph;
    Dim g_objMC as IMediaControl   ' from the General Declarations section
    ...
    Set g_objMC = New FilgraphManager   ' create the new filter graph
    ' ...
    ' Use the common File Open dialog to let the user select the input file
    CommonDialog1.ShowOpen   ' user selects a source or filter graph
    ' call IMediaControl.RenderFile to add all filters and connect all pins
    g_objMC.RenderFile CommonDialog1.filename   ' generates the complete graph
```

Creating a New Filter Graph and Adding Filters

The following code fragment demonstrates how to create the new (empty) filter graph object.

```
' fragment from the Filter Graph menu's New (empty) command handler
    Dim g_objMC as IMediaControl   ' from the General Declarations section
    ...
    Set g_objMC = New FilgraphManager   ' create the new filter graph
```

When you choose to create an empty filter graph and add individual filters, you must know the filter type. In general, there are three categories of filters: source filters, transform filters, and renderer filters. The procedure for adding source filters uses a different method than the procedure for adding transform and renderer filters.

Add source filters to the filter graph by calling the IMediaControl.AddSourceFilter method and supplying the name of a file of the specified source type or stored filter graph.

The main form of the application includes a button labeled "Add Source Filter..." that uses the common File Open dialog box to query the user for the name of the source file or stored filter graph. The application supplies the specified file as the parameter to AddSourceFilter.

```
Dim objFilter As Object  ' temporary object for valid syntax; not used

    CommonDialog1.ShowOpen  ' get the name of the source or filter graph file
    g_objMC.AddSourceFilter CommonDialog1.filename, objFilter
```

Add transform and renderer filters to the filter graph by calling the IRegFilterInfo.Filter method. The IRegFilterInfo object can be obtained from the IMediaControl.RegFilterCollection property, which represents the collection of filter objects registered with the system and available for use.

After creating the filter graph and obtaining the IMediaControl object, use the following procedure to add filters.

1. Obtain the list of registered filters by getting the IMediaControl.RegFilterCollection property.
2. Search through the collection for the desired filter. Each element in the collection is an IRegFilterInfo object.
3. Add the filter to the filter graph by calling the IRegFilterInfo.Filter method.

In the sample program, the list box labeled "Registered filters" contains the names of all the filters that appear in the RegFilterCollection property. The following code fragment illustrates steps 1 and 2 in the previous procedure.

```
 ' code fragment that populates the registered filter list box
' global variable g_objRegFilters is set to IMediaControl.RegFilterCollection
' Set g_objRegFilters = g_objMC.RegFilterCollection
Dim filter As IRegFilterInfo
    listRegFilters.Clear
    If Not g_objRegFilters Is Nothing Then
        For Each filter In g_objRegFilters  ' for each filter in collection
            listRegFilters.AddItem filter.Name ' add name to the list box
        Next filter
    End If
```

The sample application contains an Add button that adds the selected registered filter to the current filter graph. The following code fragment illustrates step 3 in the previous procedure.

```
' code fragment from the event handler for the "Add" button
Dim filter As IRegFilterInfo
' find the selected filter and add it to the graph
' g_objRegFilters is the IMediaControl object RegFilterCollection property
For Each filter In g_objRegFilters
    If filter.Name = listRegFilters.Text Then  ' the selected filter?
        Dim f As IFilterInfo  ' yes
        filter.filter f  ' add to the filter graph, return IFilterInfo f
        If f.IsFileSource Then
            CommonDialog1.ShowOpen
            f.filename = CommonDialog1.filename
        End If
        Exit For
    End If
Next filter
```

Connecting Filter Pins

After adding individual filters to the filter graph, you can establish connections between the filters by explicitly connecting each pin, or by automatically generating all connections that are needed downstream from a specific pin.

In both cases, you must traverse the hierarchy of DirectShow objects to obtain the IPinInfo object that represents a pin of the filter object. This involves finding the desired filter in the filter collection of the filter graph object, then finding the desired pin in the pin collection of the filter object.

Listing Filters in the Filter Graph

All filters in the filter graph are available in a collection that you can access using the IMediaControl.FilterCollection property.

When the user adds a filter to the filter graph, the application refreshes the list of current filters by using the IMediaControl.FilterCollection property, as shown in the following code fragment.

```
' refresh the list box that contains the current filters in the graph
    listFilters.Clear
    For Each objFI In g_objMC.FilterCollection
        listFilters.AddItem objFI.Name ' add to list box
    Next objFI
```

Listing Pins Defined for a Filter

You can access the pins defined for a filter object through the IFilterInfo.Pins property. The Pins property is a collection of individual IPinInfo objects.

After you obtain an individual IPinInfo object from the collection, you can access its properties and call its methods, as shown in the following code fragment.

```
For Each objPin In g_objSelFilter.Pins
    If objPin.Name = listPins.Text Then  ' selected pin?
        Set g_objSelPin = objPin ' yes, update information
        ' ... perform operations using that pin
    End If
Next objPin
```

After you have obtained the pin object, you can explicitly connect to one other pin or automatically generate all subsequent pin connections needed to render the pin.

Explicitly Connecting Two Pins

The IPinInfo object provides three methods to connect pins: Connect, ConnectDirect, and ConnectWithType. **Connect** adds other transform filters as needed, **ConnectDirect** does not add transform filters, and **ConnectWithType** performs the connection only if the specified pin matches the specified media type.

The sample application connects two pins using the IPinInfo.Connect method, as shown in the following code fragment. You can call the Connect method from either of the two pins that are to be connected.

```
' The sample application displays another form to select the second pin
' or "other pin" that is to be connected to this pin.
    frmSelectPin.OtherDir = g_objSelPin.Direction
    Set frmSelectPin.Graph = g_objMC  ' give that form a copy of the graph
    Set frmSelectPin.SelFilter = g_objSelFilter  ' and the current filter
    frmSelectPin.RefreshFilters  ' display available filters to connect
    frmSelectPin.Show 1  ' display the form
    If frmSelectPin.bOK Then ' user has selected one--used OK button
        Dim objPI As IPinInfo
        Set objPI = frmSelectPin.SelPin ' get the new pin from the form
        g_objSelPin.Connect objPI  ' connect the two pins
        RefreshFilters  ' display the latest pin information
    End If
```

Automatically Connecting Pins

Call the IPinInfo.Render method to automatically generate all portions of the filter graph that are needed downstream from that pin.

The term *downstream* refers to all connections needed to construct a complete path from that pin to a renderer filter. For example, consider the representation of the filter graph by the Filter Graph Editor, which shows connections as moving from the source filter at the left to the renderer filter at the right. The Render method adds all required filters and connections to the right of the specified pin.

The application includes a Connect Downstream command button. The code that handles this command automatically establishes all pin connections downstream from the specified pin object, as shown in the following code fragment.

```
' call IPinInfo.Render
' complete the graph downstream from this pin
' g_objSelPin refers to the pin selected in the list box labeled 'Pins'
    g_objSelPin.Render
```

Viewing Pin Connection Information

When you have obtained a pin object from the collection available from the IFilterInfo.Pins property of the filter object, you can list its connection and other information.

The sample application uses the IPinInfo.ConnectedTo property to obtain the pin object to which it is connected, as shown in the following code fragment.

```
' Add detailed pin information to the text box on the right
' when the user clicks on a pin in the list box on the left
  Dim strTemp As String
  On Error Resume Next
  Dim objPin As IPinInfo
  For Each objPin In g_objSelFilter.Pins
    If objPin.Name = listPins.Text Then ' selected in list box?
      Set g_objSelPin = objPin  'yes, get all information
      strTemp = ""  ' clear existing displayed pin information
      Dim objPinOther As IPinInfo
      Set objPinOther = objPin.ConnectedTo
      If Err.Number = 0 Then  ' yes, there is a connection
        strTemp = "Connected to pin: " + objPinOther.Name + " "
        Dim objPeer As IFilterInfo
        Set objPeer = objPinOther.FilterInfo
        strTemp = strTemp + " on filter: " + objPeer.Name + " "
        Dim objMTI As IMediaTypeInfo
        Set objMTI = objPin.ConnectionMediaType
        strTemp = strTemp + vbCrLf + "Media Type: " + objMTI.Type
      End If
      If objPin.Direction = 0 Then
        strTemp = strTemp + " " + vbCrLf + "Direction: Input"
      Else
        strTemp = strTemp + " " + vbCrLf + "Direction: Output"
      End If
    txtPinInfo.Text = strTemp
    End If
  Next objPin
```

Creating a Custom Graph

The sample application featured in this article is similar to the filter graph editor utility, allowing a user to create and manage any filter graph. Most applications will not provide such a general-purpose interface—they are more likely to create only the specific filter graphs needed by the application.

The sample application provides one subroutine that creates such a custom filter graph. The Options menu offers a "Build custom graph" command that calls this subroutine.

The routine that handles this command creates five filter objects and eight pin objects. The routine then connects pins by calling the IPinInfo.Connect method.

The graph connects the following filters: AVI Source, AVI Decompressor, AVI Splitter, Video Renderer, and Audio Renderer. These filters can be connected by reusing just two pin object variables. For clarity, however, each pin object is defined using a name that indicates its position in the filter graph.

The filters and pins are declared as follows:

```
Dim pSourceFilter As IFilterInfo  ' AVI Source Filter; has two pins
Dim SourceOutputPin As IPinInfo  'Source Filter output pin

Dim pAVISplitter As IFilterInfo ' AVI Splitter
Dim SplitterInPin As IPinInfo   ' AVI Splitter pin "Input"
Dim SplitterOut00Pin As IPinInfo  ' AVI Splitter pin "Stream 00"
Dim SplitterOut01Pin As IPinInfo  ' AVI Splitter pin "Stream 01"

Dim pDECFilter As IFilterInfo  ' AVI Decompressor; has two pins
Dim DECInPin As IPinInfo   'AVI Decompressor pin "XForm In"
Dim DECOutPin As IPinInfo   ' AVI Decompressor pin "XForm Out"

Dim pVidRenderer As IFilterInfo ' Video Renderer, has one pin
Dim VidRendInPin As IPinInfo  ' Video Renderer pin "Input"

Dim pAudioRenderer As IFilterInfo 'Audio Renderer, has one pin
Dim AudioRendInPin As IPinInfo ' Audio Renderer pin "Input"
```

The application adds the source filter object by calling the IMediaControl.AddSourceFilter method:

```
' create the source filter using IMediaControl.AddSourceFilter
    CommonDialog1.ShowOpen  ' get the name of the source AVI file
    g_objMC.AddSourceFilter CommonDialog1.filename, pSourceFilter
```

The application adds the other filter objects by searching for a specific name in the registered filter collection (the filter collection is available from the IMediaControl.RegFilterCollection property), and calling the IRegFilterInfo.Filter method when it finds the specific filter to add:

```
' add all non-source filters from the collection of registered filters
    Set g_objRegFilters = g_objMC.RegFilterCollection
' use the local subroutine AddFilter to find the filter named
' "AVI Decompressor" in the collection, and set the variable pDECFilter
    AddFilter "AVI Decompressor", pDECFilter
```

The AddFilter subroutine of the application loops through all the filters present in the collection. When the names match, it calls the IRegFilterInfo.Filter method to add the filter to the filter graph:

```
Private Sub AddFilter(FName As String, f As IFilterInfo)
' call IRegFilterInfo.Filter

    Set LocalRegFilters = g_objMC.RegFilterCollection
    Dim filter As IRegFilterInfo
    For Each filter In LocalRegFilters
        If filter.Name = FName Then
            filter.filter f
            Exit For
        End If
    Next filter
```

The application calls similar code for the AVI Compressor, AVI Splitter, Video Renderer, and Audio Renderer filters. After obtaining all the filter objects, the application uses the IFilterInfo.Pins property to find specific pin objects. The code loops through all pin objects in the collection, searching for the specific pin names and setting the individual pin objects when they are found, as shown in the following code fragment.

```
' Get the source filter pin we need to build the graph
For Each pPin In pSourceFilter.Pins
      Debug.Print pPin.Name
      If pPin.Name = "Output" Then
          Set SourceOutputPin = pPin
      End If
    Next pPin

'Add DEC filter
    AddFilter "AVI Decompressor", pDECFilter
    'Print out list of pins on decompressor filter
    For Each pPin In pDECFilter.Pins
      Debug.Print pPin.Name
      ' save specific pins to connect them
      If pPin.Name = "XForm In" Then
          Set DECInPin = pPin
      End If
      If pPin.Name = "XForm Out" Then
          Set DECOutPin = pPin
      End If
    Next pPin

'Add AVI Splitter
    AddFilter "AVI Splitter", pAVISplitter
    'Print out list of pins on decompressor filter
    For Each pPin In pAVISplitter.Pins
```

```
        Debug.Print pPin.Name
        ' save specific pins to connect them
        ' pin 0, pin 1
        If pPin.Name = "input pin" Then
            Set SplitterInPin = pPin
        ElseIf pPin.Name = "Stream 00" Then
            Set SplitterOut00Pin = pPin
        ElseIf pPin.Name = "Stream 01" Then
            Set SplitterOut01Pin = pPin
        End If
    Next pPin
```

After initializing the eight pin objects, it is a simple matter to call the IPinInfo.Connect method to establish the four connections between them. The following code fragment demonstrates the connection calls.

```
' connect the pins
' Note:  error handling omitted for brevity
'Connect Source video output pin to AVI splitter input pin
    SourceOutputPin.Connect SplitterInPin
' Connect AVI splitter stream 00 to AVI decompressor
    SplitterOut00Pin.Connect DECInPin
' Connect AVI splitter stream 01 to audio renderer
    SplitterOut01Pin.Connect AudioRendInPin
 'Connect AVI decompressor output pin to Video renderer input pin
    DECOutPin.Connect VidRendInPin
```

You can establish the connection from either pin; after a successful call to the Connect method, you can access the connection information from either pin object.

Summary

In summary, this article discussed the use of the following DirectShow objects, properties, and methods.

Task	**DirectShow properties or methods**
Create a new, empty filter graph.	Set objMediaControl = New FilgraphManager.
Generate the complete filter graph for a specific file.	Call the IMediaControl.RenderFile method.
Add a source filter.	Call the IMediaControl.AddSourceFilter method.
Add a renderer or transform filter.	Get the IRegFilterInfo objects using the IMediaControl.RegFilterCollection property; call the IRegFilterInfo.Filter method.
List the pins of a filter object.	Get the IPinInfo objects using the IFilterInfo.Pins property.
Explicitly connect two pins.	Call the IPinInfo.Connect method.
Create all connections from the pin to the renderer filter.	Call the IPinInfo.Render method.

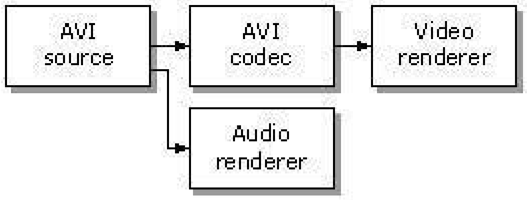
Controlling Filter Graphs Using Visual Basic

This article describes how to use the methods, events, and properties exposed by the Microsoft® DirectShow™ dynamic-link library, Quartz.dll, to render a stream of time-stamped video data in applications based in Microsoft Visual Basic®. This article is written for the Visual Basic developer who is already familiar with Windows®-based application programming, Windows-based multimedia programming, and Automation features of the Visual Basic programming system version 5.0.

Contents of this article:

DirectShow Filters and Filter Graphs

When multimedia is displayed in an application by using Quartz.dll, the application is using a collection of objects called *filters*; this collection is sometimes called a *filter graph*. The following diagram depicts a filter graph that is capable of rendering an audio-video interleaved (.avi) file.



In this illustration, the AVI source filter reads the file from disk. The AVI decompressor filter (codec) decompresses the video data as it is passed from the source filter. The codec filter then passes this data to the video renderer. The video renderer, in turn, passes the data to the device in a format that the device understands. The AVI source filter passes the audio data directly to the audio renderer, which, in turn, passes the data to the audio device.

In addition to filters, DirectShow supports an Automation object called the *filter graph manager*. This object knows about the available filters and understands which filter types are required to render which file formats. The filter graph manager exposes the methods, events, and properties supported by the filters in a given graph. The filter graph manager also exposes its own set of methods, events, and properties. These are exposed by using interfaces, which are simply collections of related methods, events, and properties.

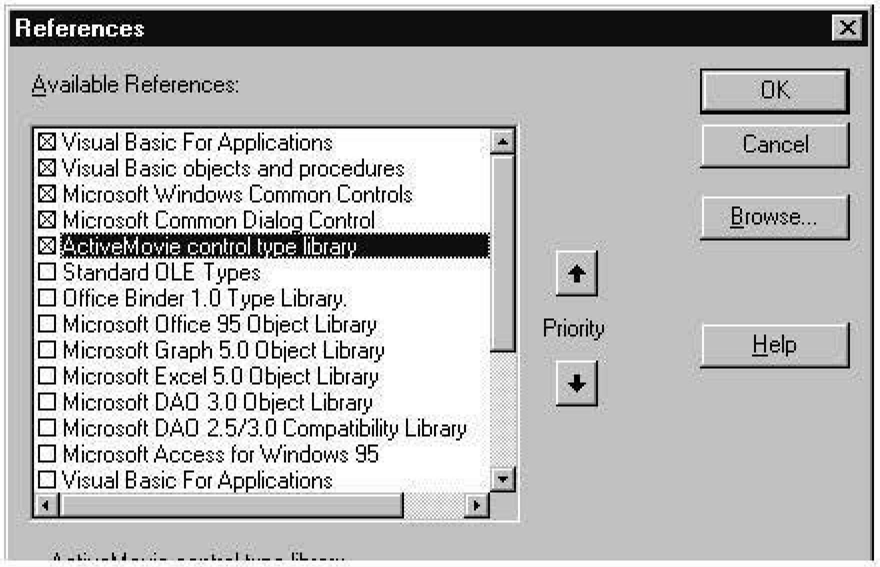
The following table identifies the DirectShow interfaces available in Quartz.dll for use with Visual Basic-based applications, and describes the purpose of each interface.

Interface	**Description**
IAMCollection	Accesses pin and filter collections.
IBasicAudio	Controls and retrieves current volume setting.
IBasicVideo	Controls a generic video renderer.
IFilterInfo	Retrieves information about a filter and about pin objects in the filter.
IMediaControl	Instantiates the filter graph and controls media flow (running, paused, stopped).
IMediaEvent	Allows customized event handling for events such as repainting, user termination, completion, and so on.
IMediaPosition	Controls and retrieves start time, stop time, rate, and current position.
IMediaTypeInfo	Retrieves the media type and subtype.
IPinInfo	Accesses pin information, such as pin direction and media type, and controls pin connection, disconnection, and rendering.
IRegFilterInfo	Contains information about registered (transform and render) filters.
IVideoWindow	Controls window aspects of a video renderer.

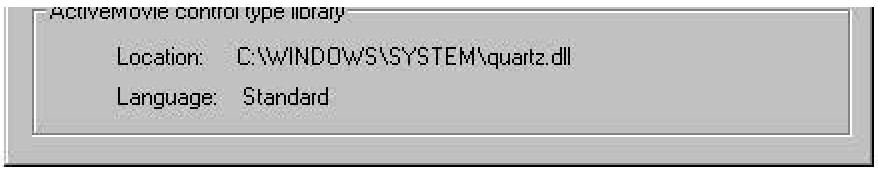
DirectShow Interfaces, Visual Basic Objects, and Registering Quartz.dll

To use the DirectShow interfaces in your Microsoft Visual Basic-based application, you must register the ActiveMovie Control type library in your Visual Basic project.

When you register the ActiveMovie Control type library by using the Visual Basic References dialog box, you are identifying the type library that contains the Automation information that Visual Basic requires. The following illustration shows the References dialog box.



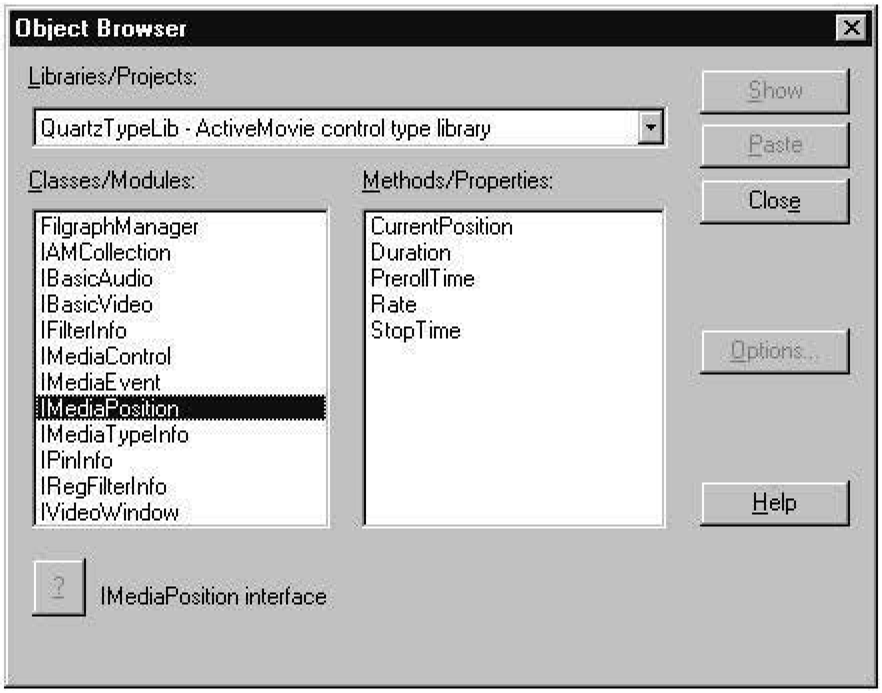
☒ Visual Basic For Applications
☒ Visual Basic objects and procedures
☒ Microsoft Windows Common Controls
☒ Microsoft Common Dialog Control
☒ ActiveMovie control type library
☐ Standard OLE Types
☐ Office Binder 1.0 Type Library.
☐ Microsoft Office 95 Object Library
☐ Microsoft Graph 5.0 Object Library
☐ Microsoft Excel 5.0 Object Library
☐ Microsoft DAO 3.0 Object Library
☐ Microsoft DAO 2.5/3.0 Compatibility Library
☐ Microsoft Access for Windows 95
☐ Visual Basic For Applications


ActiveMovie control type library

Location: C:\WINDOWS\SYSTEM\quartz.dll

Language: Standard

Once the type library is registered, you can use the Object Browser dialog box to view the list of methods, events, and properties associated with a given interface.



Classes/Modules:

- FilgraphManager
- IAMCollection
- IBasicAudio
- IBasicVideo
- IFilterInfo
- IMediaControl
- IMediaEvent
- IMediaPosition
- IMediaTypeInfo
- IPinInfo
- IRegFilterInfo
- IVideoWindow

Methods/Properties:

- CurrentPosition
- Duration
- PrerollTime
- Rate
- StopTime

IMediaPosition interface

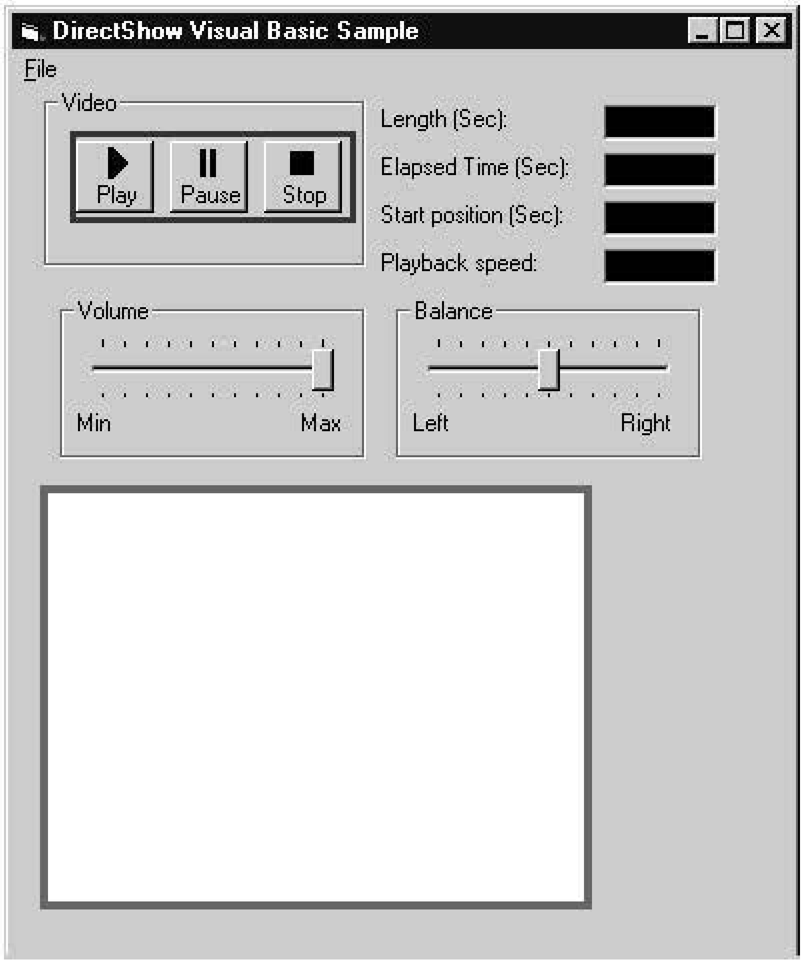
Note The type information in the filter graph manager is organized by interface, rather than object.

The VBDemo Sample Application

This section is based on the VBDemo sample application, which enables the user to do the following:

- Choose an DirectShow file (.avi, .mpg, or .mov).
- Display a simple toolbar that lets the user play, pause, or stop the rendering.
- Display the length of the video and the elapsed time.
- Display start position and run rate.
- Display a volume control and a balance control.
- Position the destination window (a shape control) below all other controls in the main form.

The application, when running a file, appears as follows:



Installing the Files

Before using the DirectShow objects in your Visual Basic-based application, you must install Quartz.dll in the Windows\System directory and ensure that the appropriate entries are made in your system's registry database. Currently, the DirectShow Software Development Kit (SDK) setup program automates this process. To install, start Setup.exe and choose the Runtime option. The dynamic-link library (DLL) will be copied to the correct location, and the registry will be automatically updated.

Registering Quartz.dll with Visual Basic

Open the Visual Basic application and choose the References command from the Tools menu to verify that the files were installed succesfully. (At startup, Visual Basic examines the registry database for registered automation controls and adds their names to the list that appears in this dialog box.) To use the filter graph manager, click the box that appears next to the ActiveMovie Control type library name.

Once Visual Basic registers the type information, you can use the filter graph manager and its associated interfaces in your application.

Preparing to Use the DirectShow Objects

Visual Basic initializes all objects in the VBDemo sample program using the FilgraphManager object, which implements the following interfaces.

- IBasicAudio
- IBasicVideo
- IMediaControl
- IMediaEvent
- IMediaPosition
- IVideoWindow

Each of the interfaces is accessed by a Visual Basic programmable object defined to be of that interface type. The objects in the sample application are defined as global variables in the general declarations section, as shown in the following example.

```
Dim g_objVideoWindow As IVideoWindow           'VideoWindow Object
Dim g_objMediaControl As IMediaControl         'MediaControl Object
Dim g_objMediaPosition As IMediaPosition       'MediaPosition Object
Dim g_objBasicAudio  As IBasicAudio            'Basic Audio Object
Dim g_objBasicVideo As IBasicVideo             'Basic Video Object
```

All the programmable objects are initialized using FilgraphManager, as shown in the following example:

```
     Set g_objMediaControl = New FilgraphManager
    g_objMediaControl.RenderFile (g_strFileName)     ' name of input file
    ...
    Set g_objBasicAudio = g_objMediaControl
    Set g_objVideoWindow = g_objMediaControl
    Set g_objMediaEvent = g_objMediaControl
    Set g_objMediaPosition = g_objMediaControl
```

The other interfaces available for use with Visual Basic-based applications are obtained by calling methods that explicitly return the desired interface. The following table summarizes how to obtain these interfaces.

Interface	**Methods that return the interface pointer**
IAMCollection	IPinInfo.MediaTypes, IFilterInfo.Pins, IMediaControl.FilterCollection, IMediaControl.RegFilterCollection
IFilterInfo	First IMediaControl.FilterCollection, then IAMCollection.Item or IPinInfo.FilterInfo
IMediaTypeInfo	IPinInfo.ConnectionMediaType
IPinInfo	IFilterInfo.FindPin, IAMCollection.Item
IRegFilterInfo	First IMediaControl.RegFilterCollection, then IAMCollection.Item

For a sample that shows how to manipulate these filter and pin interfaces, see Constructing Filter Graphs Using Visual Basic.

Instantiating the Filter Graph

You can use the filter graph manager to render existing files of the following types.

- .avi (audio-video interleaved)
- .mov (Apple® QuickTime®)
- .mpg (Motion Picture Experts Group)

In addition, you can use the filter graph manager to render an existing filter graph by specifying the file that contains that graph as a parameter to the RenderFile method.

Because the filters in a filter graph are dependent on the type of file being rendered, the sample application does not instantiate a filter graph until the user selects a file. The code that handles this selection is embedded in the procedure that opens the file, mnu_FileOpen. This code displays the Show Open common dialog box and stores the selected file name in a *g_strFileName* variable. After this, the code verifies that the correct file type was chosen. Quartz.dll issues an error message if it is passed a file extension other than .mpg, .avi, or .mov.

Once the *g_strFileName* variable is set, the application instantiates the filter graph manager and creates the filter graph object. The filter graph manager is instantiated when the Visual Basic keyword *New* is used to create the AUTOMATION object. The filter graph object is created when the IMediaControl::RenderFile method is called, as shown in the following example.

```
    'Instantiate a filter graph for the requested
    'file format.

    Set g_objMediaControl = New FileGraphManager
    g_objMediaControl.RenderFile (g_strFileName)
```

Rendering Video

The IMediaControl interface supports three methods (Run, Pause, and Stop) that an application can call to render, pause, or stop a video stream. After the filter graph object is instantiated, your application can call these methods.

The sample application displays a toolbar from which the user controls video rendering. When the user clicks Run, the Run method is activated and a global Boolean variable (*fVideoRun*) is set to True. This variable is used in a timer procedure that retrieves the current media position (or elapsed rendering time). If the Pause or Stop button is clicked, this variable is set to False, and the current media position is not retrieved during timer events.

The code that activates the Run, Pause, and Stop methods is found in the Toolbar1_ButtonClick procedure. The toolbar contains buttons that are numbered 1, 3, and 5; the buttons numbered 2 and 4 are separators that provide additional space between the buttons, as shown in the following example.

```
Private Sub Toolbar1_ButtonClick(ByVal Button As Button)
' handle buttons on the toolbar
' buttons 1, 3 and 5 are defined; 2 and 4 are separators
' all DirectShow objects are defined only if the user
' has already selected a filename and initialized the objects

    ' if the objects aren't defined, avoid errors
    If g_objMediaControl Is Nothing Then
        Exit Sub
    End If

    If Button.Index = 1 Then 'PLAY
        'Invoke the MediaControl Run() method
        'and play the video through the predefined
        'filter graph.

        g_objMediaControl.Run
        g_fVideoRun = True

    ElseIf Button.Index = 3 Then  'PAUSE
        'Invoke the MediaControl Pause() method
        'and pause the video that is being
        'displayed through the predefined
        'filter graph.

        g_objMediaControl.Pause
        g_fVideoRun = False

    ElseIf Button.Index = 5 Then  'STOP

        'Invoke the MediaControl Stop() method
        'and stop the video that is being
        'displayed through the predefined
        'filter graph.

        g_objMediaControl.Stop
        g_fVideoRun = False
        ' reset to the beginning of the video
        g_objMediaPosition.CurrentPosition = 0
        txtElapsed.Text = "0.0"
```

Controlling Audio

The IBasicAudio interface supports two properties: the Volume property and the Balance property. The **Volume** property retrieves or sets the volume. In the sample application, this property is bound to the slider control slVolume. The **Balance** property retrieves or sets the stereo balance. In the sample application, this property is bound to the slider control slBalance.

Note The volume is a linear volume scale, so only the far right side of the slider is useful.

The following example shows the code that adjusts the volume (by setting the g_objBasicAudio.Volume property) is found in the slVolume_Change procedure.

```
Private Sub slVolume_Change()

    'Set the volume on the slider

    If Not g_objMediaControl Is Nothing Then
    'if g_objMediaControl has been assigned

       g_objBasicAudio.Volume = slVolume.Value

    End If

End Sub
```

Scaling and Translating the Video Output

The IVideoWindow interface supports the methods and properties you can use to alter the size, state, owner, palette, visibility, and so on, for the destination window. If you are not concerned with the location or appearance of the destination window, you can render output in the default window (which appears in the upper-left corner of the desktop) without calling any of these methods or properties.

The sample application moves the destination window to a position below the other controls on its main form. In addition to moving the window, the sample application alters the window style by removing the caption, border, and dialog box frame. To do this, set the g_objVideoWindow.WindowStyle property to 0x06000000. This corresponds to the logical OR operation of the values WS_DLGFRAME (0x04000000) and WS_VSCROLL (0x02000000). For a complete list of window styles and corresponding values, see the Winuser.h file in the Microsoft® Platform SDK.

To move the destination window onto the form, specify a new position by setting the Top and Left properties of g_objVideoWindow. The **Top** and **Left** properties are set to correspond to the upper-left corner of a blank control with a rectangular shape, a placeholder of sorts, that appears on the form. The ScaleMode property for the form was set to 3, which specifies units of pixels. This allows the form properties and DirectShow object properties to be used without conversion. The DirectShow object properties are also expressed in pixels. The default units for a Visual Basic form are twips.

The sample application uses the left top of the placeholder rectangle, then resizes the shape based on the size of the specified video. The application determines the required size of the rectangle by retrieving the source video width and height. These values correspond to the VideoWidth and VideoHeight properties of the g_objBasicVideo object.

In addition to setting the Top and Left properties, it is necessary to identify the form of the application as the new parent window by passing the window handle of the form, *hWnd*, to the g_objVideoWindow.Owner property. If the handle is not passed, the destination window will appear relative to the desktop and not the form.

The following example shows the tasks that are accomplished in the mnu_FileOpen procedure.

```
    Set g_objVideoWindow = g_objMediaControl
    g_objVideoWindow.WindowStyle = CLng(&H6000000)    ' WS_DLGFRAME | WS_VSCROLL
    g_objVideoWindow.Left = CLng(Shape1.Left)    ' shape is a placeholder on the for
    g_objVideoWindow.Top = CLng(Shape1.Top)
    Shape1.Width = g_objVideoWindow.Width    ' resize the shape given the input vid
    Shape1.Height = g_objVideoWindow.Height
    g_objVideoWindow.Owner = frmMain.hWnd    ' set the form as the parent
```

The following example shows how the **ScaleMode** property is initialized in the Form_Load procedure.

```
' code fragment from the Form_Load procedure
' ...
    frmMain.ScaleMode = 3    ' pixels
' ...
```

Avoid attempting to scale the destination window by setting the Width and Height properties for the g_objVideoWindow object, because performance suffers considerably.

Tracking Status

The g_objMediaPosition object exposes a number of properties that you can use to retrieve or set the current position, stop point, duration, and rate. When the user selects a file, the sample application retrieves and displays the duration, starting position, and rate. The corresponding code appears in the mnu_FileOpen procedure, as shown in the following example.

```
    Set g_objMediaPosition = g_objMediaControl
    g_dblRunLength = g_objMediaPosition.Duration
    txtDuration.Text = CStr(g_dblRunLength)    ' display the duration
    g_dblStartPosition = 0.0
    txtStart.Text = CDbl(g_dblStartPosition)    ' display the start time
    g_dblRate = g_objMediaPosition.Rate
    txtRate.Text = CStr(g_dblRate)
```

The current position is also displayed, using a timer that is started when the user clicks Run. When the user clicks Run, a global Boolean variable, *g_fVideoRun*, is set to True, indicating that the program should retrieve and display the current media position, which is expressed as the elapsed rendering time from the absolute beginning of the multimedia stream.

If Pause or Stop is clicked, the variable is set to False, and the current media position is not retrieved. The corresponding code (which displays the current position) appears in the Timer1_Timer procedure, as shown in the following example.

```
Private Sub Timer1_Timer()
    'Retrieve the Elapsed Time and
    'display it in the corresponding
    'textbox.
Dim Dbl As Double

    If g_fVideoRun = True Then
        Dbl = g_objMediaPosition.CurrentPosition
        If Dbl < g_dblRunLength Then
            txtElapsed.Text = CStr(Dbl)
        Else
            txtElapsed.Text = CStr(g_dblRunLength)
        End If
    End If
End Sub
```

Getting and Setting the Start Position

The sample application uses the IMediaPosition.CurrentPosition property to let the user adjust the point at which the video begins rendering. If the user enters a new CurrentPosition and then clicks Play, the video begins rendering at the frame whose timestamp is closest to the requested time.

In addition to adjusting the starting time, the user can jump to new frames while the video is rendering by specifying a new value in the corresponding text box and then pressing ENTER.

The code that handles the CurrentPosition property is found in the following example of the txtStart_KeyDown procedure.

```
 Private Sub txtStart_KeyDown(KeyCode As Integer, Shift As Integer)
' handle user input to change the start position
If KeyCode = vbKeyReturn Then
    If g_objMediaPosition Is Nothing Then
        Exit Sub
    ElseIf CDbl(txtStart.Text) > g_dblRunLength Then
        MsgBox "Specified position invalid: re-enter new position."
    ElseIf CDbl(txtStart.Text) < 0 Then
        MsgBox "Specified position invalid: re-enter new position."
    ElseIf CDbl(txtStart.Text) <> "" Then
        g_dblStartPosition = CDbl(txtStart.Text)
        g_objMediaPosition.CurrentPosition = g_dblStartPosition
    End If
End If
End Sub
```

Getting and Setting the Rate

The sample application uses the IMediaPosition.Rate property to let the user adjust the rate at which the video is rendered. This rate is a ratio with respect to typical playback speed. For example, a rate of 0.5 causes the video to be rendered at one-half its typical speed, and a rate of 2.0 causes the video to be rendered at twice its typical speed.

Unlike the CurrentPosition property, which can be set while the video is being rendered, the Rate property must be set prior to rendering.

Note The sound track can be disabled for some videos when the rate is less than 1.0.

The code that handles the Rate property is found in the following txtRate_KeyDown procedure.

```
Private Sub txtRate_KeyDown(KeyCode As Integer, Shift As Integer)
' DirectShow VB sample
' handle user updates to the Rate value
If KeyCode = vbKeyReturn Then
    If g_objMediaPosition Is Nothing Then
        Exit Sub
    ElseIf CDbl(txtRate.Text) < 0# Then
        MsgBox "Negative values invalid: re-enter value between 0 and 2.0"
    ElseIf CStr(txtRate.Text) <> "" Then
        g_dblRate = CDbl(txtRate.Text)
        g_objMediaPosition.Rate = g_dblRate
    End If
End If
End Sub
```

Cleaning Up

Each time your application uses the Visual Basic Set statement to instantiate a new DirectShow object, it must include a corresponding Set statement to remove that object (and its corresponding resources) from memory prior to shutdown. For example, in the mnu_FileOpen procedure, a new g_objBasicAudio object is instantiated with the following syntax.

```
Set g_objBasicAudio = g_objMediaControl
```

When the user selects Exit from the File menu, a corresponding Set statement removes this object:

```
Set g_objBasicAudio = Nothing
```

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

List of Filters and Samples

Microsoft® DirectShow™ provides filters and samples as part of the DirectShow Software Development Kit (SDK). The filters are supplied as binary code only and are listed among the filters available in the Filter Graph Editor when you choose **Insert Filters** from the **Graph** menu. A sample filter includes source code and you must build and register it before it will appear in the Filter Graph Editor. In addition to sample filters, the SDK contains sample applications that demonstrate how to use filters.

See the following sections for a list of filters, sample filters, and sample applications supplied with DirectShow:

- Filters
- Sample Filters
- Sample Applications

Filters

The DirectShow SDK provides the following filters:

- ACM Audio Compressor
- Analog Video Crossbar
- Audio Capture
- Audio Renderer
- AVI Compressor
- AVI Decompressor
- AVI Draw
- AVI MUX
- AVI Splitter
- AVI/WAV File Source
- Color Space Converter
- Cutlist File Source
- DSound Audio Renderer
- DV Muxer
- DV Splitter
- DV Video Decoder
- DV Video Encoder
- DVD Navigator
- File Source (Async)
- File Source (URL)
- File Stream Renderer
- File Writer
- Full Screen Renderer
- Indeo 4.3 Video Compression
- Indeo 4.3 Video Decompression
- Indeo 5.0 Audio Decompression
- Indeo 5.0 Video Progressive Download Sources
- Indeo 5.0 Video Compression
- Indeo 5.0 Video Decompression
- Internal Script Command Renderer

- Line 21 Decoder
- Lyric Parser
- MIDI Parser
- MIDI Renderer
- MPEG Audio Decoder
- MPEG Video Decoder
- MPEG-1 Stream Splitter
- Multi-File Parser
- Overlay Mixer
- QuickTime Decompressor
- QuickTime Movie Parser
- SAMI (CC) Parser
- TrueMotion 2.0 Decompressor
- TV Audio
- TV Tuner
- VFW Video Capture
- VGA 16 Color Ditherer
- Video Renderer
- WAVE Parser
- WDM Video Capture

Sample Filters

The DirectShow SDK provides the following sample filters:

- Async Sample (Asynchronous Reader Filter)
- Ball Sample (Bouncing Ball Filter)
- Contrast Sample (Video Contrast Filter)
- Dump Sample (Dump Filter)
- EzRGB24 Sample (Image Effect Filter)
- Gargle Sample (Gargle Filter)
- Inftee Sample (Infinite-Pin Tee Filter)
- MPGAudio Sample (MPEG Audio Decoder Filter)
- MPGVideo Sample (MPEG Video Decoder Filter)
- Nullip Sample (Null In Place Filter)
- Nullnull Sample (Minimal Null Filter)
- Scope Sample (Oscilloscope Filter)
- Synth Sample (Audio Synthesizer Filter)
- SampVid Sample (Video Renderer Filter)
- TextOut Sample (Text Display Filter)
- Vcrctrl Sample (VCR Control Filter)
- VidCap Sample (Video Capture Filter)

Sample Applications

The DirectShow SDK provides the following sample applications:

- AMCap Sample (DirectShow Capture Application)
- CLText Sample (Text Cutlist Application)
- CPlay Sample (C/COM-based Media Player Application)

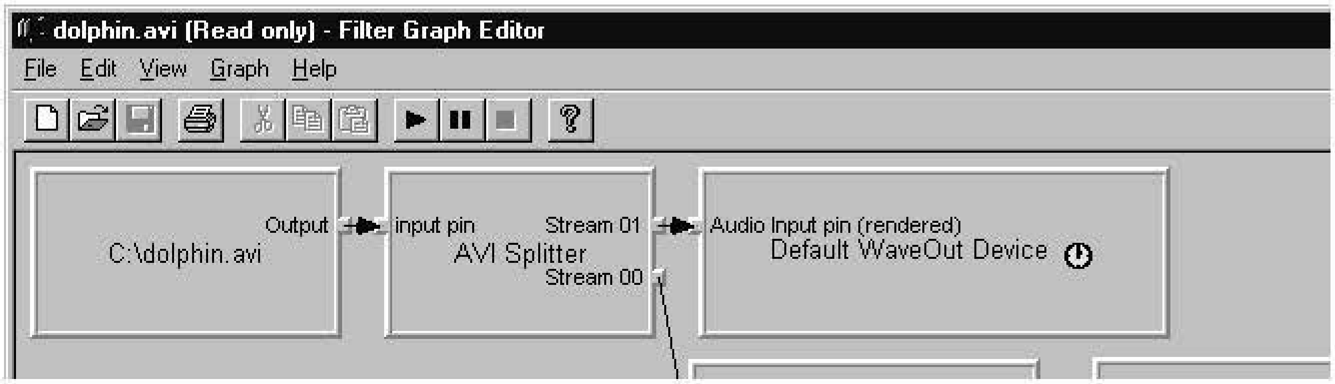
- Dvdsampl Sample (DVD Player Application)
- InWindow Sample (Window Playback Application)
- IPlay Sample (Indeo Player Application)
- MFCPlay Sample (C++/COM-based Media Player Application)
- MPEGProp Sample (MPEG Property Page Display Application)
- PlayFile Sample (Simple Playback Application)
- PID Sample (Plug-In Distributor Application)
- ShowStrm Sample (Multimedia Streaming Application)
- Simpleci Sample (Cutlist Application)
- VidClip Sample (Video Editing Application)
- Visual Basic-Based ActiveX Player
- Visual Basic-Based Filter Graph Builder
- Visual Basic-Based Filter Graph Player
- Visual Basic-Based Player

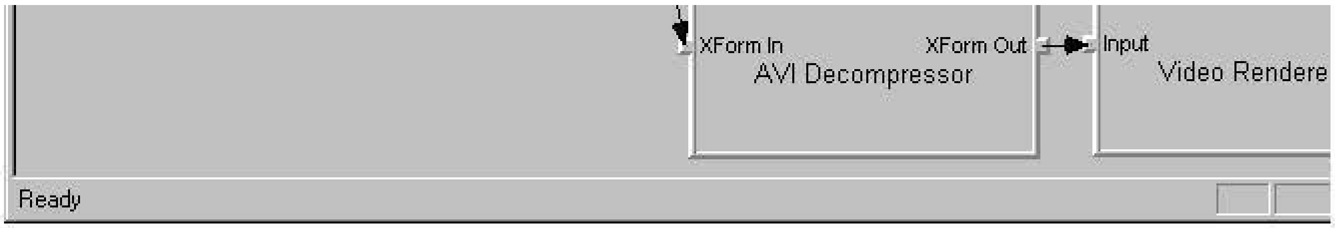
Previous Home Topic Contents Index Next

About the DirectShow Filter Graph Editor

The Microsoft® DirectShow™ SDK provides the Filter Graph Editor tool (also referred to as "Graphedt.exe" or "GraphEdit") that you can use to create, edit, and save filter graphs. This article introduces GraphEdit, and discusses the purpose of the buttons on the GraphEdit toolbar in About the GraphEdit Toolbar. See Using the Filter Graph Editor for detailed information about how to use GraphEdit.

The DirectShow architecture uses filter graphs to manage multimedia streams in Microsoft Windows® 95 and Microsoft Windows NT®. A filter graph consists of a set of filters connected in sequence; the sequence typically includes a source filter reading from a media file, a transform filter, and a renderer, although your graph will vary depending on its purpose. A graph to play a media file with video and audio typically includes a source filter, a stream splitter, a video decompression filter, and appropriate renderers. The following screen shot shows a filter graph in GraphEdit that includes both audio and video streams.

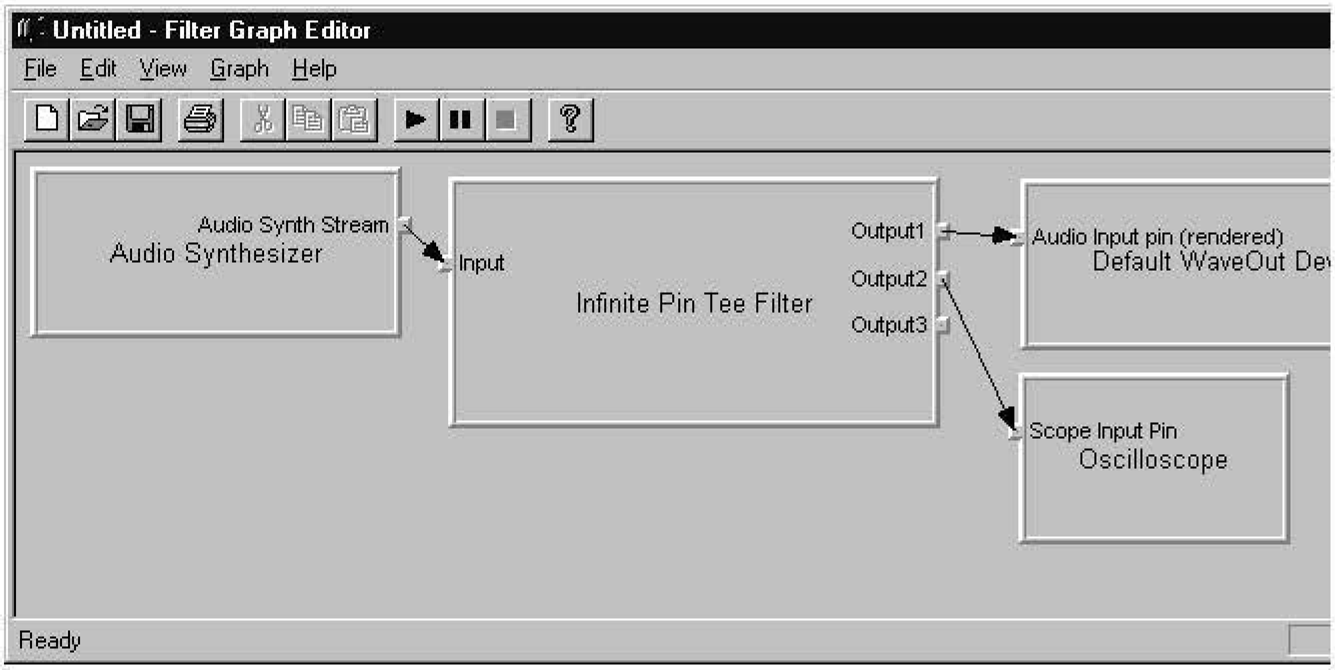

dolphin.avi (Read only) - Filter Graph Editor
File Edit View Graph Help
C:\dolphin.avi
Output
input pin
AVI Splitter
Stream 01
Stream 00
Audio Input pin (rendered)
Default WaveOut Device



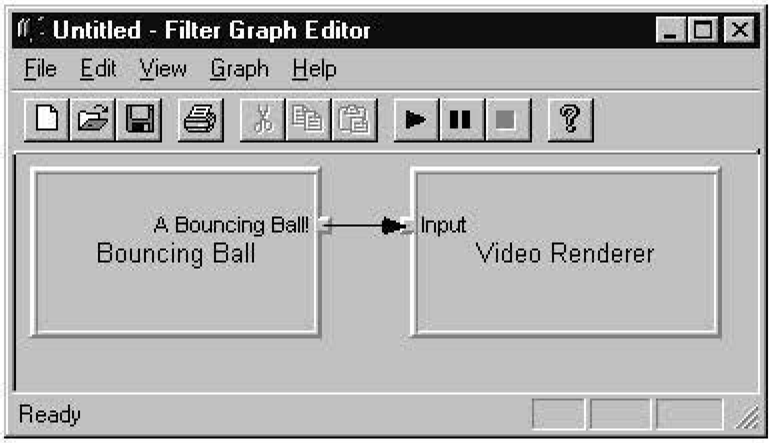
GraphEdit helps you visualize a filter graph as shown in the preceding screen shot. A rectangle represents a filter, with tiny attached squares representing pins. Pins are key to understanding DirectShow, from format negotiation to data transportation. Arrows joining the pins represent data flow paths, much like a pipe directs the flow of water. GraphEdit displays input pins on the left side of the filter's rectangle and output pins on the right side. Output pins might not be present until you connect an input pin. Input pins only connect to output pins, and vice versa.

GraphEdit indicates the audio renderer that provides the clock for the filter graph by a small yellow-filled clock symbol within the audio renderer's rectangle. You can disable the clock by canceling the selection of **Use Clock** on the **Graph** menu. Graphs without clocks play the audio and video streams as fast as possible, independently of each other.

The following filter graph plays back audio from the Audio Synthesizer source filter (Synth). Synth generates its own audio data rather than reading data from a file. The audio renderer's clock is enabled in this filter graph.



The following filter graph is one of the simplest possible filter graphs. The Bouncing Ball source filter generates video data and the video renderer filter displays it.



About the GraphEdit Toolbar

GraphEdit's toolbar appears beneath the menu bar. It provides shortcut commands for opening new filter graphs, saving the current filter graph, and playing, pausing, or stopping the multimedia source file. You can display or hide the toolbar by clicking **Toolbar** on the **View** menu.

Button	**Effect**
🗋	Creates a new, empty filter graph.
📂	Opens an existing multimedia source file or an existing filter graph (.grf) file.
💾	Saves the current filter graph as a filter graph (.grf) file.
🖨	Prints the current filter graph.
▶	Plays the multimedia source using the current filter graph.
❚❚	Pauses play of the multimedia source.
■	Stops play of the multimedia source.
?	Displays Help information for GraphEdit.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

Using the Filter Graph Editor

This article steps through how to open and use the Filter Graph Editor (GraphEdit) to create filter graphs and to play them back. See About the DirectShow Filter Graph Editor for an introduction to GraphEdit.

Contents of this article:

Starting GraphEdit

Start GraphEdit in one of the following ways.

- Click **Start**, and then point to **Programs**. On the DirectX Media SDK version 5.*x* submenu, click **GraphEdit**.
- Open the DXMedia\Tools folder (assuming the default installation directory for the DirectX media SDK) and double-click the Graphedt icon.
- Drag and drop a multimedia file, such as .avi or .mpg, onto the Graphedt icon.

Creating a New Filter Graph

This section discusses ways to create a new filter graph using GraphEdit. It contains the following topics:

Drag Files Onto GraphEdit

If GraphEdit is not running, you can drag a multimedia file, such as .avi or .mpg, onto the Graphedt icon and GraphEdit will run and automatically build the filter graph for the media file.

If GraphEdit is already running, drag a multimedia file into its client area to have it automatically build the filter graph.

Use the Open Command

Choose the **Open** command from the GraphEdit **File** menu, or choose the **Open** button from the GraphEdit toolbar, to open a media file or saved filter graph file (.grf). GraphEdit automatically generates the complete filter graph.

Use the Render Media File Command

You can use the GraphEdit **Render Media File** command to automatically generate the complete filter graph for a multimedia source.

To generate a complete filter graph for a given source file perform the following steps:

1. On the **File** menu, click **Render Media File**.
2. From the "Select a file to be rendered" dialog box, choose a multimedia source file, such

as a .avi, .mpg, or .wav file.

3. Click **Open**.

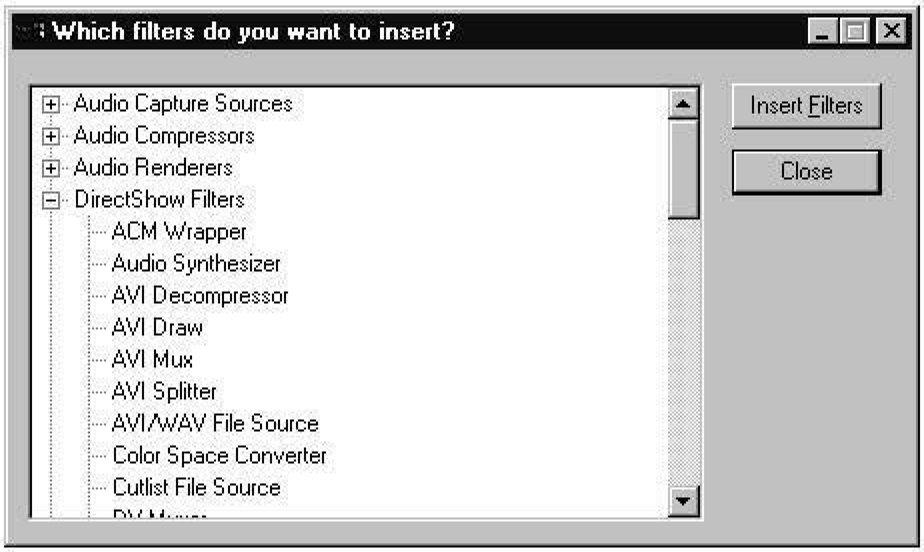
GraphEdit adds and connects all filters needed to render the source file automatically.

Manually Build a Filter Graph

To create an empty filter graph and manually add filters and connections, perform the following steps. This example assumes you want to play a file from your hard disk.

See About the DirectShow Filter Graph Editor for information about how GraphEdit represents filters, pins, and connections graphically.

1. On the **File** menu, click **New** to create a new filter graph.
2. On the **Graph** menu, click **Insert Filters**. GraphEdit displays the "Which filters do you want to insert?" dialog box, which contains a list of filter categories.
3. Click the plus symbol (+) immediately to the left of the DirectShow Filters category to see the drop-down list of filters. The plus symbol becomes a minus symbol (–) when the list is expanded, as the following illustration shows. Click the minus symbol to contract the list.



4. Select File Source (Async) from the filter list (scrolling if necessary) and click the **Insert Filters** button. Because the Asynchronous File Source filter requires an input file, GraphEdit displays the "Select an input file for this filter to use" dialog box.
5. Select a multimedia file that you have on your hard disk (for this example, assume you chose a file called Jupiter.avi). Click the **Open** button. GraphEdit inserts a rectangle labeled Jupiter.avi in its client area. This rectangle represents the Asynchronous File Source filter and has a small square attached to its right side, labeled Output, which represents the filter's output pin.
6. At this point, you could right-click the Async filter's output pin and choose **Render** from the resulting shortcut menu to have GraphEdit render the rest of the filter graph for you automatically. Instead, continue by inserting a few more filters manually. From the "Which filters do you want to insert?" dialog box, select the AVI Splitter filter. Like the File Source (Async) filter, this filter is listed in the DirectShow Filters category. Click **Insert Filters** and GraphEdit inserts the AVI Splitter filter in its client area. This filter has one input pin, shown by a small square attached to the left side of the AVI Splitter filter rectangle.

7. Connect the Async File Source filter to the AVI Splitter filter as follows.
   1. Click the Async filter's Output pin and drag to the AVI Splitter's input pin. GraphEdit creates an arrow representing the connection between the filters, and moves the arrow in response to the dragging operation.
   2. Release the mouse button when the tip of the arrow head is over the small square representing the AVI Splitter's input pin. The interior of the pin's square turns black when the arrow head is in a valid location. After you've connected these filters, the AVI Splitter filter sprouts one output pin for each stream in the file. If Jupiter.avi contains an audio stream and a video stream, the pins will be labeled Stream 01 and Stream 00.
8. At this point, you could right-click each of the AVI Splitter's output pins in turn and choose **Render** to have GraphEdit render the rest of the filter graph for you automatically. Instead, continue by inserting one more filter manually. From the "Which filters do you want to insert?" dialog box, open the Audio Renderers filter category by clicking on the plus symbol (+) immediately to the left of the Audio Renderers label. Select a default audio renderer (for example, the Default WaveOut Device or Default DirectSound Device if available) and click the **Insert Filters** button.
9. Connect the AVI Splitter's Stream 01 output pin by dragging from that pin and releasing the mouse button when the arrowhead is over the input pin of the audio renderer.
10. You could continue to insert filters manually, much in the same way that you have so far. Instead, right-click the AVI Splitter's Stream 00 output pin and choose **Render** to have GraphEdit build up the rest of the filter graph for you automatically.

Running and Editing a Filter Graph

After you've built a filter graph as outlined in Creating a New Filter Graph, you can play or pause the filter graph. To do so, select **Play** or **Pause** from the GraphEdit **Graph** menu.

Play plays the filter graph. If the graph includes the video renderer filter, any video data (such as a movie), plays in the video renderer window. If the filter graph includes an audio renderer, any sound associated with the movie plays as well. Pause cues up data in the filter graph and displays the first frame of video data, enabling playback to happen quickly if you later select Play. When the filter graph is playing or paused, you can select **Stop** from the **Graph** menu to stop playback.

You can also play, pause, or stop the filter graph by choosing the appropriate buttons from the GraphEdit toolbar. See About the GraphEdit Toolbar for more information.

You can edit the filter graph when it is stopped. Select either a filter or connection between filters by clicking the filter or connecting arrow. GraphEdit highlights the object by placing a blue border around it. You can highlight multiple objects at once by clicking outside an object and dragging diagonally to create a selection rectangle. When you release the mouse button, objects contained within the selection rectangle are highlighted. Press the DELETE key to delete a highlighted object or group of objects. Insert new filters by choosing **Insert Filters** from the **Graph** menu, and make new connections as outlined in Manually Build a Filter Graph.

You can drag filters in the GraphEdit client area if you want to reposition them. You might want to do this to make the filters fit on one screen without scrolling.

Viewing Properties in GraphEdit

GraphEdit enables you to view the properties of filters, pins, and connections. To view the properties of an object such as a filter, right-click the filter and choose **Properties** from the resulting shortcut menu. The options provided by a property sheet vary depending on the filter, pin, or connection. Typically, the property sheet for a filter includes tabs for its pins.

COM Overview

The Component Object Model (COM) is a binary standard that defines how objects are created and destroyed and, most importantly, how they interact with each other. As long as applications follow the COM standard, different applications from different sources can communicate with each other across process boundaries. People use COM to make communication with other applications easy.

Because COM is a binary standard, it is language independent. You do not have to use C++ to implement COM. You can use any language that supports tables of function pointers.

A COM *interface* is a collection of logically related methods that express a single functionality. For example, the IAsyncReader interface enables reading of MEDIATYPE_Stream data. All COM interfaces derive from IUnknown, and all are named by a globally unique interface identifier (IID).

A COM *class* is an implementation of one or more COM interfaces, and a COM object is an instance of a COM class. A Microsoft® DirectShow™ filter, for example, is a COM object. Each object has a globally unique class identifier (CLSID).

Globally unique identifiers (GUIDs) are extremely long integers that identify COM interfaces and objects, and are used to eliminate name collisions across applications.

All access to a COM object is through pointers to its interfaces. Interface methods are purely virtual and are stored in a table called a *vtable*. The interface pointer points to the vtable's beginning. A COM interface defines the parameter types and the syntax for each of its methods. The COM class provides an implementation for each method of the interface.

Once a COM class has been defined and assigned a CLSID, you can create an instance of the object. There are several ways to create an instance of the class, including using the COM CoCreateInstance or IClassFactory::CreateInstance methods , or the C++ **new** operator.

When you create an instance of an object, the call returns a pointer to one of the object's interfaces. Once you have an initial pointer to an interface on the object, you can use the IUnknown::QueryInterface method to find out whether the object supports another specific interface, and, if so, to get a pointer to that interface. COM supplies many standard interfaces that support data storage and transfer, notification, and basic connectivity with other objects, including IStream, IPropertyPage, and IMoniker. DirectShow, in turn, adds its own COM interfaces, such as IAMDirectSound, that clients of DirectShow objects can query for to determine if the object supports a particular functionality. To use COM interfaces, clients must know the interface definitions and the IID to query for (IID_*interfacename*). For example, assume you have a pointer to a COM object's IUnknown interface in the *pUnknown* variable. You can query to see if the object supports **IAMDirectSound** with the following code.

```
hr=pUnknown->QueryInterface(IID_IAMDirectSound, (void **)&pIAMDSound);
```

IUnknown is the basic COM interface on which all others are based. **IUnknown** has three methods—QueryInterface, AddRef, and Release—that implement interface querying and reference counting. All COM interfaces inherit these three methods from **IUnknown**.

Reference counting is the technique by which an object (or, strictly, an interface) decides when it is no longer being used and can therefore destroy itself. COM objects are dynamically allocated from within the object and multiple clients can use them simultaneously. To avoid wasting memory, the COM object must keep track of the number of clients using it, and destroy itself when clients no longer need it. The number of clients using the object is maintained in the reference count. Every time a new interface pointer to the COM object is created, the client using the object must increase the reference count by calling AddRef on the interface pointer. Every time a client destroys an interface pointer to the object, it must first decrease the reference count by calling Release on the interface pointer.

Binding associates a method with a pointer to its memory location. At compile time, a COM object's client is bound to the vtable locations of the object's interface methods. This is called *early binding*. With some languages, such as Microsoft® Visual Basic®, a vtable interface is difficult to access. Dispatch interfaces, identified by dispatch identifiers (DISPIDs), allow clients to access member functions not by position in a vtable, but by a human-readable name. Dispatch interfaces are accessed through the COM IDispatch interface and its **Invoke** method, which converts the names of the dispatch interface's functions to DISPIDs. The client retrieves the DISPIDs at run time. This is called *late binding*. To allow late binding, a COM object must implement the **IDispatch** interface and a mapping of function names and function parameters to a set of DISPIDs. In DirectShow, CBaseDispatch implements the **IDispatch** interface.

Marshaling is the process of passing function arguments and return values among processes and machines. An in-process proxy packages arguments for the member function of an object in another process, and generates a remote procedure call to the other process. In the other process, a stub receives the call and unpacks the data, and calls the object through its interface. Dispatch interfaces do not need proxies and stubs and so are easier to use than vtable interfaces in out-of-process applications. Vtable interfaces, however, can be considerably faster, particularly in in-process applications. You can also write dual interfaces that have both tables of function pointers and dispatch interfaces. Dual interfaces can be nearly as fast as vtable interfaces, while allowing the flexibility of dispatch interfaces.

For more information about how DirectShow uses COM, see DirectShow and COM. For general information about COM, see the "COM" section in the Microsoft Platform SDK, or an introductory book such as *ActiveX OLE* by David Chappell.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

Overview of DVD Interfaces and Data Types

This article provides an overview of the DVD interfaces and data types used in Microsoft® DirectShow™.

Contents of this article:

DVD Application-Level Interfaces

The following list shows DVD interfaces that media developers can use to create applications.

IAMLine21Decoder
: Provides access to closed captioning information and settings. Closed captioning information is transmitted in the vertical blanking interval (VBI) of television signals, specifically on line 21 (Line21) of field 1 in the VBI.

IDvdControl
: Controls the playback and search mechanisms of a DVD-Video disc that contains one or more video movies.

IDvdGraphBuilder
: Enables the DVD application writer to easily build a filter graph for DVD-Video playback.

IDvdInfo
: Enables an application to query for attributes of available DVD-Video titles and the DVD player status. This interface also allows for control of a DVD player beyond Annex J in the DVD specification.

IMixerPinConfig
: Exposed on the input pins of overlay mixer filters and contains methods that manipulate video streams in various ways. The overlay mixer filter contains multiple input pins that are dynamically created as video input streams are added.

DVD Decoder Filter Interfaces

The following list shows DVD interfaces that developers can use to set and retrieve device and sample properties.

IKsPropertySet
: Enables you to set and retrieve device properties. Use this interface to set and retrieve any of the properties from the following list.
  - DVD Copy Protection Property Set — These properties provide authentication of copy protection information from hardware or software decrypters.
  - DVD Subpicture Property Set — These properties control the color, contrast, and output of the subpicture display.
  - DVD Time Stamp Rate Change Property Set — These properties enable you to change DVD playback rate, by modifying timestamps between input and output pins on two filters.

IMediaSample2
: Enables you to set and to retrieve sample properties. This interface is derived from the

IMediaSample interface and uses the following data types:

- AM_MEDIA_TYPE structure — Describes a media sample type. This structure can include the following substructures.
    - VIDEOINFOHEADER2 structure — Describes the bitmap and color information for a video image, including interlace, copy protection, and pixel aspect ratio information.
    - MPEG2VIDEOINFO structure — Describes an MPEG-2 video stream.
- AM_SAMPLE_PROPERTY_FLAGS enumerated type — Indicates values for the **dwSampleFlags** member of the AM_SAMPLE2_PROPERTIES structure.
- AM_SAMPLE2_PROPERTIES structure — Generic media sample properties structure.

IVPConfig
: Enables a video port (VP) mixer filter to communicate with a VP driver (decoder), to set and retrieve configuration information. This interface assumes that the mixer filter creates the video port.

IVPNotify
: Enables you to control the properties of a filter that uses a video port. This interface derives from the IVPBaseNotify interface.

DVD Filter Graph Media Types, Formats, and Events

The following articles provide more information about DVD:

- DirectShow DVD Support — Provides a diagram of a DVD filter graph and outlines the media types and data formats used in each connection.
- DVD Event Notification Codes — Describes DirectShow system-defined events, which filters in the filter graph pass to the filter graph manager. Filters pass these events to the filter graph manager by using the IMediaEventSink::Notify method, and the application retrieves them with the IMediaEvent::GetEvent method.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

About WDM Video Capture

This article provides an overview of video capture using the Microsoft® Windows® 98 and Windows NT® Driver Model (WDM) and Microsoft DirectShow™. It describes the close association between WDM video capture and DirectShow.

This article also briefly describes the close association between the Stream class and WDM Connection and Streaming Architecture (CSA) and video capture minidrivers (which are clients of the Stream class driver). However, you should have a basic understanding of these topics before reading this article. For background information, see http://www.microsoft.com/hwdev/pcfuture/.

Contents of this article:

A Brief History of Windows Video Capture

Microsoft released Video for Windows 1.*x* in November 1992 for Windows 3.1 and optimized it for capturing movies to disk. Since then, video capture rates have risen dramatically due to use of the PCI bus, bus mastering controllers, NT striped sets, Fast/Wide SCSI, and direct transfer of captured video from adapter memory to disk without data copies. Despite capture rates that now exceed 20 MB per second, and a large number of clients for Video for Windows, deficiencies in the Video for Windows architecture exposed by the emergence of video conferencing required development of a new video capture technology.

The Video for Windows architecture lacks features important to video conferencing, television viewing, capture of video fields, and additional data streams such as vertical blanking interval (VBI). Vendors have extended Video for Windows by implementing proprietary extensions, which are product specific, to circumvent these limitations. However, without standardized interfaces, applications that use these features must include hardware-dependent code. The tight coupling between Video for Windows capture drivers and display drivers means that changes made to capture drivers require changes to display drivers as well.

In addition, the Video for Windows interface, **AVICap**, doesn't work well with DirectShow because **AVICap** allocates buffers. If DirectShow is accessed through **AVICap**, the buffers must be copied at the transition point, which is very inefficient. With the integration of digital versatile disc (DVD), MPEG decoders, video decoders and tuners, video port extensions (VPE), and audio codecs on single adapters, a unified driver model that supports all of these devices and handles resource contention is needed.

WDM Overview

DirectShow supplies backward compatibility for Video for Windows applications without the shortcomings of Video for Windows. WDM video capture aims to provide additional support for the following: USB conferencing cameras, 1394 DV devices, desktop cameras, TV viewing, multiple video streams support and VPE capture support. This support is provided through kernel-based streaming.

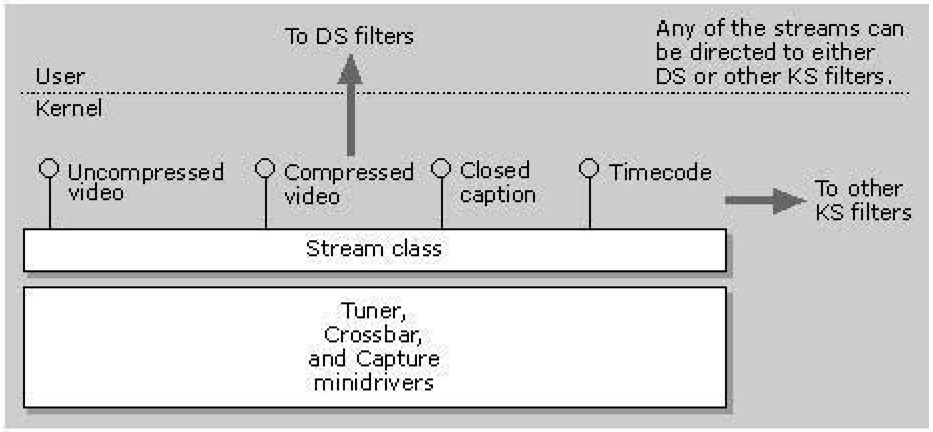
WDM-based streaming extends the nonkernel streaming of DirectShow by providing a kernel connection. Streaming services are processed by the WDM Streaming Class Driver and other cross-process system software components. The WDM Streaming Class Driver is also responsible for calling a minidriver, which is a hardware-specific dynamic-link library (DLL) provided by IHVs to support device-specific controls. The minidriver and the Microsoft-provided WDM Streaming Class Driver work together to provide low-level services for the lowest latency streaming, and DirectShow provides the higher-level features specific for your application. Because the Stream class supports a uniform streaming model for standard and custom data types, and supports data transfer between kernel drivers without requiring a transition to user mode, it is a highly efficient mode to use.

Previously, DirectShow filters transmitted data to and from the kernel whenever necessary to achieve things like decompression or rendering. Unfortunately, each of these transitions of the data stream from user mode to kernel mode was time-consuming because of the transition itself, and because of parameter validation, security validation, and possibly data copying, that must occur.

Through kernel streaming, a stream makes fewer transitions between user and the kernel mode. It can be either partially or entirely produced and consumed in kernel mode. When streams are processed in kernel mode, the DirectShow filters merely expose control mechanisms to direct the streams. This greatly reduces the overhead associated with numerous transitions between modes.

Kernel streams can pass data to the filter graph at appropriate points, depending on the application. The following diagram illustrates the transition to user mode.

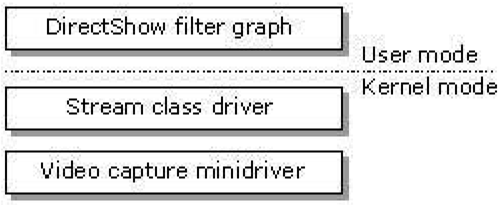
Kernel Streaming



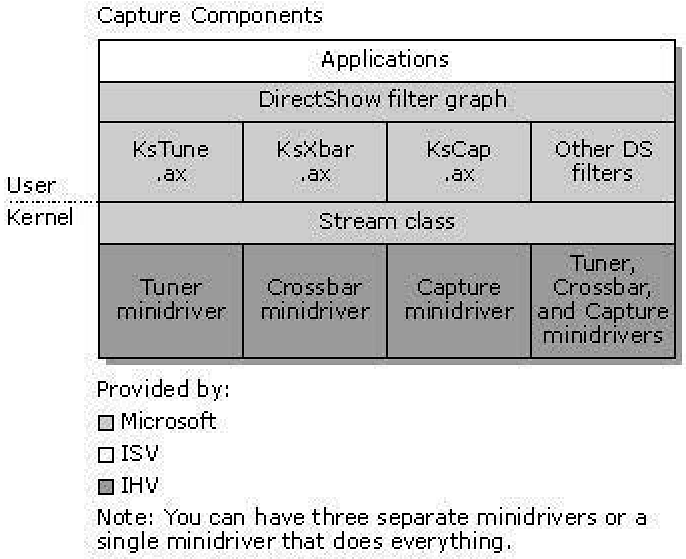
During video capture, the stream class passes uncompressed video data back to the video capture filter for writing or rendering. Also, because kernel streaming supports multiple streams, other types of data contained in the stream, such as timecode or closed captioning, could be passed up simultaneously.

WDM Video Capture Architecture

The following diagram shows the three basic components of the WDM capture architecture.



Because the WDM capture architecture was designed to integrate smoothly with DirectShow, it is straightforward to build capture graphs in DirectShow by using WDM capture filters that send control messages from DirectShow into the streaming class. The Ksproxy.ax, Kstune.ax, and Ksxbar.ax filters, which are scheduled to ship in the Windows 98 Device Driver Kit (DDK), enable WDM streaming data, such as data from Universal Serial Bus (USB) conferencing cameras, 1394 DV devices, TV viewing, and desktop cameras, to be easily controlled and sent by the Stream class to the DirectShow capture graph. The following diagram demonstrates how these components are integrated into the basic architecture.



In this diagram, the Ksproxy.ax, Kstune.ax, Ksxbar.ax, and other DirectShow filters communicate directly with the Stream class. The Microsoft WDM Stream class, through its participation in CSA, transports high-bandwidth, time-stamped, latency-sensitive data streams between kernel mode components or between kernel mode drivers and user-mode components. Through CSA, the Stream class works well with DirectShow in that it shares media types, has similar streaming states (Stop, Pause, and Run), and shares the same concept of pins and connections. This provides an easy transition of data from the Stream class to the filters in the filter graph.

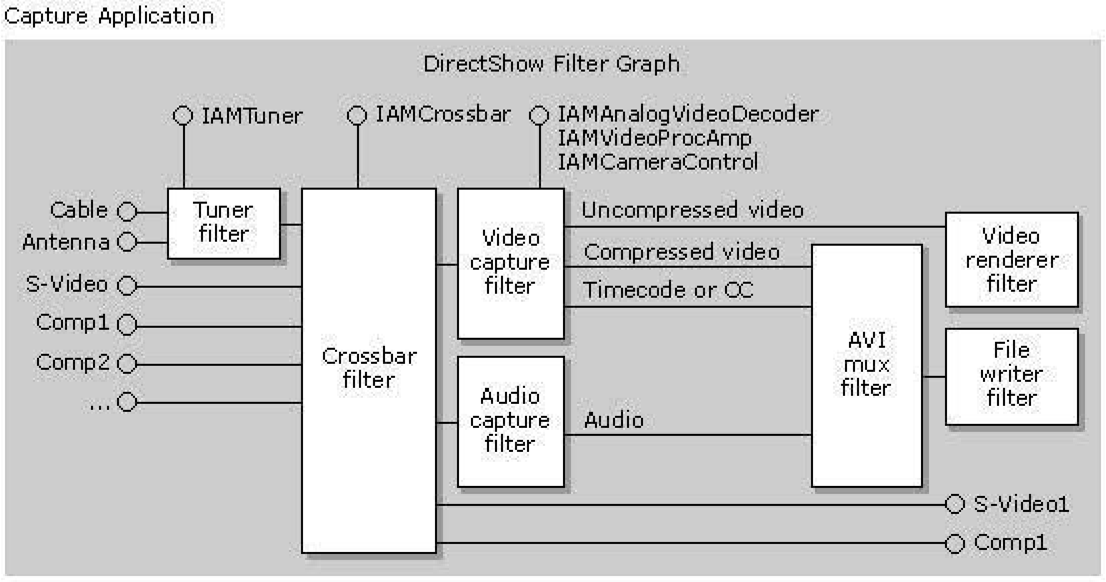
The Ksproxy.ax, Kstune.ax, and Ksxbar.ax filters also have supporting minidrivers (or one minidriver that supports all three). Video capture minidrivers are clients of the Stream class and control hardware devices that produce streams of video images and related data. These minidrivers provide the following functionality:

- Capture of compressed and uncompressed video streams, vertical blanking interval data, timecode, and ancillary data streams.
- Control of devices associated with video streams such as TV tuners, video routing devices, TV audio control, and video compressors.
- Compatibility with WDM-CSA.

Stream class video capture drivers can support multiple, simultaneous streams of compressed and uncompressed video, timecode, closed caption, raw and decoded VBI data, as well as custom data formats. For each data type that can be produced simultaneously with other data types, the driver should create a new stream. The Stream class exposes each stream as a separate WDM-CSA pin. Each stream can be connected to another WDM-CSA kernel filter, or it can make the transition to user mode and flow on an output pin of a DirectShow user-mode filter. Each stream (or pin) can support a variety of different formats. For example, a single pin can provide RGB16, RGB24, YVU9, and JPEG digital video. For more information on minidrivers, see the Windows 98 DDK.

Filter Graph Configuration

The association between DirectShow filters and CSA makes DirectShow filters a powerful and relatively safe method for manipulating data from a kernel mode Stream class driver. The flexibility of DirectShow makes numerous configurations of filters possible. The following diagram shows one possible configuration of user-mode DirectShow filters for simultaneous preview and capture of video to disk.



In this diagram, an incoming TV signal can be tuned in with the Tuner filter and routed with the Crossbar filter. The filter graph passes data in various streams to the video or audio capture filters to be saved on disk. This includes audio streams, video streams, and any other ancillary data in various streams such as SMPTE timecode or closed captioning data.

Conclusion

WDM video capture was designed to resolve the problems inherent in the Video for Windows architecture. The main advantages of WDM video capture are:

- 32 bit drivers.
- Synergy between DirectShow and CSA.
- Single class driver architecture for hardware (such as video ports and chip sets) that is shared between video capture devices and DVD/MPEG devices.
- Television tuner, input selection, and support for fields, VBI, and video port extensions (VPE).
- One driver works on both Windows 98 and NT platforms.

Because of the large installed base of Video for Windows applications, Video for Windows drivers will continue to be used for devices that are primarily used for capturing movies. Version 1.1e of Video for Windows currently ships in Windows 98 to provide operating system support for these devices. However, the WDM video capture architecture provides optimal support for capture devices used primarily for TV viewing and video conferencing.

New WDM Capture Interfaces and Filters

Some of the new WDM capture interfaces exposed by the kernel streaming filters are IAMTVTuner, IAMCrossbar, **IAMAnalogVideoEncoder**, **IAMAnalogVideoDecoder**, IAMVideoProcAmp, and **IAMCameraControl**.

Some of the new Windows 98 video capture filters for WDM are Ksproxy.ax, Kstune.ax, Ksxbar.ax. See the Windows 98 DDK for more information on these filters.

Application Developer's Guide

If you are creating an application that uses DirectShow components, read the articles in this section. These articles pertain to writing applications in both C and Microsoft® Visual Basic®.

How to...

This section gives step-by-step procedures for creating applications, adding features to applications, and registering DirectShow objects. Topics include how to play a movie from C++, how to build an application in Visual C++, how to display a filter's property page, and how to use multimedia streaming, control the video playback window, or enumerate hardware devices from an application.

Play a Movie from C++

This article walks through a simple C++ program designed to demonstrate one way to play movies. It is based on the PlayFile function code, taken from the Playfile.cpp file, which you can find in the Playfile sample in the Samples\ds\player directory.

The PlayFile function has no control over the filters selected by the filter graph manager to play the input media file or over the playback window created. This article contains the following sections.

- Playing a Media File — the basic code to play back a media file.
- Adding Media Seeking — shows the code needed to seek to a particular location in the media file.

See these related sections to add a particular feature to your playback code:

- Control the Video Playback Window from C++ — demonstrates how to set the playback window's style and position.
- Display a Filter's Property Page from C++ — demonstrates how to display a filter's property page, so that the user can change how media files are played back.

Playing a Media File

This section explains the code needed to play a media file from within C/C++. The Playfile sample contains Playfile.cpp and demonstrates how to create an application window, display a menu to open a media file, and call the PlayFile function to play the media file. You can examine the Playfile application in the Samples\ds\player\Playfile directory to see how to use the PlayFile function. To learn how to build the Playfile sample from Visual C++ 5.*x*, see Setting the Visual Studio Include and Lib Directories.

The PlayFile function plays a specified file in a playback window. This function uses the filter graph manager to automatically render the media clip. The filter graph manager selects the appropriate filters and constructs the filter graph.

PlayFile function code demonstrates:

- Basic interfaces needed to play and control a media file.
- Instantiating the filter graph manager.
- Calling the filter graph manager to build the filter graph that renders the media file.
- Playing the media file.
- Accessing events to tell when the playback is finished (media file ended).

After any function call that retrieves an interface pointer (CoCreateInstance, RenderFile, and QueryInterface), you should insert error-checking code to make sure the interface pointer was successfully obtained; if it wasn't, release any interfaces pointers already obtained. An example of error-checking code is:

```
if (FAILED(hr)) {
    goto ObjectRelease; // go to the clean-up section
}
```

You can call the PlayFile function from an application with code such as the following:

```
TCHAR *szFilename = "c:\\dxmedia\\movie\\movie.avi";
PlayFile(szFilename);
```

Perform the following steps to play a media file from within C/C++. You don't necessarily have to perform the steps in the order presented.

1. Include the necessary headers.

```
#include <windows.h>
#include <mmsystem.h>
#include <streams.h>
#include "playfile.h"
```

2. Define a windows message constant and the HELPER_RELEASE macro, which will be used to release the interfaces from the WndMainProc callback (see the Playfile code for generic window code).

```
#define WM_GRAPHNOTIFY  WM_USER+13
#define HELPER_RELEASE(x) { if (x) x->Release(); x = NULL; }
```

3. Declare variables.

```
HWND      ghApp;
HINSTANCE ghInst;
HRESULT   hr;
LONG      evCode;
LONG      evParam1;
LONG      evParam2;
```

The *ghApp* variable is the handle of window to notify when the graph signals an event. The *ghInst* variable is the HINSTANCE of the window. The *evCode* variable will hold the event code, and the *evParam1* and *evParam2* variables will hold the event parameters.

4. Declare and initialize the necessary interfaces. The reference count of the interfaces is automatically incremented on initialization, so you don't need to call the IUnknown::AddRef method on them. For this example, you need only the four interfaces shown in the following code. For more information, see the documentation for the IMediaEventEx, IGraphBuilder, IMediaControl, and IVideoWindow interfaces.

```
IGraphBuilder *pigb  = NULL;
IMediaControl *pimc  = NULL;
IMediaEventEx *pimex = NULL;
IVideoWindow  *pivw  = NULL;
```

5. Define the function. The *szFile* parameter is the name of the media file that will be played.

```
void PlayFile (LPSTR szFile)
{
    HRESULT hr;
```

6. Create a Unicode (wide character) string from the input file name.

```
WCHAR wFile[MAX_PATH];
MultiByteToWideChar( CP_ACP, 0, szFile, -1, wFile, MAX_PATH );
```

7. Instantiate the filter graph manager, asking for the IGraphBuilder interface.

```
hr = CoCreateInstance(CLSID_FilterGraph,
NULL,
CLSCTX_INPROC_SERVER,
IID_IGraphBuilder,
(void **)&pigb);
```

8. Query for the IMediaControl interface (provides the methods to run, pause, and stop the playback), the IMediaEventEx interface (so you can receive event notifications), and the IVideoWindow interface to hide the window when the movie is finished playing.

```
pigb->QueryInterface(IID_IMediaControl, (void **)&pimc);
pigb->QueryInterface(IID_IMediaEventEx, (void **)&pimex);
pigb->QueryInterface(IID_IVideoWindow, (void **)&pivw);
```

9. Ask the filter graph manager to build the filter graph that renders the input file. This does not play the media file. (When you play the file with **Run**, the filter graph will automatically render the input file's media type. You do not have to specify a renderer filter.)

```
hr = pigb->RenderFile(wFile, NULL);
```

10. Use a window to capture graph signal events. This not only improves performance, but allows your application to run in any threading model.

```
pimex->SetNotifyWindow((OAHWND)ghApp, WM_GRAPHNOTIFY, 0);
```

    The window specified by *ghApp* will handle messages in response to all events from the graph. If an event occurs, DirectShow posts a WM_GRAPHNOTIFY message to the window.

11. Start playing the media file.

```
hr = pimc->Run();
```

    Alternatively, if your playback had a pause or stop button (see, for example, the CPlay sample or Controlling Filter Graphs Using C), you can pause or stop the playback on the button event with the IMediaControl::Pause or IMediaControl::Stop method, as shown in the following code.

```
        hr = pimc->Pause();
        hr = pimc->Stop();
```

The WndMainProc callback function in Playfile handles the filter graph messages and releases the interfaces when necessary, using the HELPER_RELEASE macro. The GetClipFileName function gets the movie to be played, while the WinMain function creates the window. These are all generic windows functions.

This section showed how to play a media file from the beginning. The next section shows how to control where within a media file to start and stop playing.

Adding Media Seeking

You can use the IMediaPosition or IMediaSeeking interface to seek to a particular place in your media file. The IMediaPosition::put_CurrentPosition method enables you to specify a start time within the media file. For example, you can use the following code to rewind to the media file's beginning.

```
IMediaPosition *pimp;
hr = pigb->QueryInterface(&IID_IMediaPosition, (void **)&pimp);
hr = pimp->put_CurrentPosition(0);
```

Time is specified in 100-nanosecond units. The following code seeks into the media file 1 second:

```
  hr = pimp->put_CurrentPosition(10000000);
```

You can use the IMediaPosition::put_StopTime method to set the time within the media file to stop playback.

However, with IMediaPosition you can seek only to times within a media file. With the

IMediaSeeking interface, you can set your seeking time format to 100-nanosecond time units, frames, bytes of data, media samples, or interlaced video fields. You set the format you want to use with the IMediaSeeking::SetTimeFormat method. Make sure your media file is not playing when you the set the format.

The term *media time* refers to positions within a seekable medium. Media time can be expressed in a variety of units, and indicates a position within the data in the file. The following table shows the possible media time formats.

Value	**Description**
TIME_FORMAT_MEDIA_TIME	Seeks to the specified time in the media file, in 100-nanosecond units. This is the default.
TIME_FORMAT_BYTE	Seeks to the specified byte in the stream.
TIME_FORMAT_FIELD	Seeks to the specified interlaced video field.
TIME_FORMAT_FRAME	Seeks to the specified video frame.
TIME_FORMAT_SAMPLE	Seeks to the specified sample in the stream.

For example, the following code sets the format so that the application seeks for sample numbers.

```
IMediaSeeking *pims;
hr = pigb->QueryInterface(IID_IMediaSeeking, (void **)&pims);
hr = pims->SetTimeFormat(&TIME_FORMAT_SAMPLE);
```

An application can use the various seeking modes to seek in a stream to a particular video frame or audio sample without doing time/rate conversions itself. This is useful for editing, which requires sample-accurate playback. The frame or sample number that the application specifies is passed through to the AVI or MPEG parser without the risk of rounding errors.

The following steps show how to set which frame in a media file to start playing at and which frame to stop playing at; for example, to start playing a movie at the fifth frame after its beginning. You can insert this code into the PlayFile function anywhere after the RenderFile function has built the filter graph.

1. Access the IMediaSeeking interface.

   ```
   IMediaSeeking *pims;
   hr = pigb->QueryInterface(IID_IMediaSeeking, (void **)&pims);
   ```

2. Set the time format. In the following example, the time format is set to seek to frames.

   ```
   hr = pims->SetTimeFormat(&TIME_FORMAT_FRAME);
   ```

3. Declare and initialize the media-seeking variables. In this case, they are frames within the media file to start and stop playback. The following values set the start frame to 5 and the stop frame to 15.

   ```
   LONGLONG start = 5L;
   LONGLONG stop = 15L;
   ```

4. Set the start and stop media time with the IMediaSeeking::SetPositions method. The

AM_SEEKING_AbsolutePositioning flag means that the start and stop frames are absolute positions within the media file (not relative to the present position in the media file). In this example, the media file will start playing at frame 5 into the file and stop at frame 15, for a duration of 10 frames. The length of playing time depends on the video's frame rate.

```
pims->SetPositions(&start, AM_SEEKING_AbsolutePositioning, &stop,
            AM_SEEKING_AbsolutePositioning);
```

5. Release the IMediaSeeking interface.

```
pims->Release();
```

By removing the SetTimeFormat call and setting the values of start and stop as follows, you can set the media file to start playing 5 seconds into the file and stop 7 seconds into the file, for a duration of 2 seconds.

```
LONGLONG start = 50000000L;
LONGLONG stop = 70000000L;
```

By setting other formats in the SetTimeFormat call, you can seek to frames, sample numbers, byte, and so on.

Control the Video Playback Window from C++

This article walks through a simple C++ program designed to demonstrate one way to play movies in a particular playback window. It is based on the PlayMovieInWindow function code taken from the InWindow.cpp file, which is available in the InWindow sample in the Samples\ds\player directory. This function is based on the Playfile sample, but has been expanded to show how an application can control the size and style of the video playback window.

See these related sections if you want to play back media files or display a property page:

- Play a Movie from C++ — demonstrates the basic code for playing back a media file.
- Adding Media Seeking — shows the code needed to seek to a particular location in the media file.
- Display a Filter's Property Page from C++ — demonstrates how to display a filter's property page, so the user can change how media files are played back.

Perform the following steps to play a video file in a particular window from within C/C++. You don't necessarily have to perform the steps in the order presented.

1. Include the necessary headers.

```
#include <windows.h>
#include <mmsystem.h>
#include <streams.h>
#include "inwindow.h"
```

2. Define a windows message constant and the HELPER_RELEASE macro, which will be used to release the interfaces from the WndMainProc callback (see the InWindow code for generic window code).

```
#define WM_GRAPHNOTIFY  WM_USER+13
#define HELPER_RELEASE(x) { if (x) x->Release(); x = NULL; }
```

3. Declare variables.

```
HWND      ghApp;
HINSTANCE ghInst;
HRESULT   hr;
LONG      evCode;
LONG      evParam1;
LONG      evParam2;
RECT      grc;
```

The *ghApp* variable is the handle of window to notify when the graph signals an event. The *ghInst* variable is the HINSTANCE of the window. The *evCode* variable will hold the event code, and the *evParam1* and *evParam2* variables will hold the event parameters. The *grc* variable will hold the coordinates of the parent window's client area.

4. Declare and initialize the necessary interfaces. The reference count of the interfaces is automatically incremented on initialization, so you don't need to call the IUnknown::AddRef method on them. For this example, you need only the four interfaces shown in the following code. For more information, see the documentation for the IMediaEventEx, IGraphBuilder, IMediaControl, and IVideoWindow interfaces.

```
IGraphBuilder *pigb  = NULL;
IMediaControl *pimc  = NULL;
IMediaEventEx *pimex = NULL;
IVideoWindow  *pivw  = NULL;
```

5. Define the function and declare variables. The *szFile* parameter is the name of the video file that will be played.

```
void PlayMovieInWindow (LPCTSTR szFile)
    {
```

6. Create a Unicode (wide character) string from the input file name.

```
        WCHAR wFile[MAX_PATH];
        MultiByteToWideChar( CP_ACP, 0, szFile, -1, wFile, MAX_PATH );
```

7. Instantiate the filter graph manager, asking for the IGraphBuilder interface.

```
        hr = CoCreateInstance(CLSID_FilterGraph,
        NULL,
        CLSCTX_INPROC_SERVER,
        IID_IGraphBuilder,
        (void **)&pigb);
```

8. Query for the IMediaControl interface (provides the methods to run, pause, and stop the playback), the IMediaEventEx interface (so you can receive event notifications), and the IVideoWindow interface to hide the window when the movie is finished playing.

```
    pigb->QueryInterface(IID_IMediaControl, (void **)&pimc);
    pigb->QueryInterface(IID_IMediaEventEx, (void **)&pimex);
    pigb->QueryInterface(IID_IVideoWindow, (void **)&pivw);
```

9. Ask the filter graph manager to build the filter graph that renders the input file. This does not play the media file. (When you play the file with **Run**, the filter graph will automatically render the input file's media type. You do not have to specify a renderer filter.)

```
    hr = pigb->RenderFile(wFile, NULL);
```

10. Set the ownership of the playback window. This sets *ghApp* as the owning parent.

```
    pivw->put_Owner((OAHWND)ghApp);
```

11. Set the style of the video window. This step is very important, and you must specify the WS_CHILD, WS_CLIPCHILDREN, and WS_CLIPSIBLINGS flags.

```
    pivw->put_WindowStyle(WS_CHILD | WS_CLIPCHILDREN | WS_CLIPSIBLINGS);
```

12. Get the coordinates of the parent window's client area.

```
    GetClientRect(ghApp, &grc);
```

13. Set the playback window's position within parent's client area. In this case, the playback window fills the client area. If the video being played is smaller than the playback window it will be stretched to fit the window. If the video is larger, it will be compressed to fit the window.

```
    pivw->SetWindowPosition(grc.left, grc.top, grc.right, grc.bottom);
```

14. Start playing the media file.

```
        hr = pimc->Run();
```

The InWindow sample uses the same GetClipFileName function to get the movie to be played and the same the WinMain function to create the window as the Playfile sample.

The InWindow WndMainProc callback function is similar to the Playfile WndMainProc used to handle the filter graph messages and release the interfaces when necessary, with one important difference. The WndMainProc function in InWindow calls the IVideoWindow::put_Owner method with a NULL value for its parameter. You must do this before releasing the IGraphBuilder interface and before the video window is destroyed. Otherwise, messages will continue to be sent to the video playback window but it will have no parent to forward the messages to, so errors will likely occur.

```
pivw->put_Owner(NULL);
```

Display a Filter's Property Page from C++

This article walks through a simple C++ program designed to demonstrate how to request a filter to display its property page. It is based on the MPegProp function code in the MPegProp.cpp file, which you can find in the MPegProp sample in the Samples\ds\player directory.

This code displays the property page of the MPEG video decoder. This filter has a property page that enables you to play MPEG files in color or monochrome. You can see this filter's property page by opening the Filter Graph Editor, choosing the **Insert Filters** command from the **Graph** menu, selecting **MPEG Video Decoder** from the **DirectShow Filters** list, and clicking the **Insert Filters** button. After you've inserted the filter, right-click anywhere on it to display its property page.

See these related sections if you want to play back media files or specify the video playback window:

- Play a Movie from C++ — demonstrates the basic code for playing back a media file.
- Adding Media Seeking — shows the code needed to seek to a particular location in the media file.
- Control the Video Playback Window from C++ — demonstrates how to display a filter's property page, so the user can change how media files are played back.

Perform the following steps to control the MPEG video decoder's property page in C/C++. You don't necessarily have to perform the steps in the order presented.

1. Include the necessary headers.

```
#include <windows.h>
#include <mmsystem.h>
#include <streams.h>
#include "playfile.h"
```

2. Define a windows message constant and the HELPER_RELEASE macro, which will be used to release the interfaces from the WndMainProc callback (see the MPegProp code for generic window code).

```
#define WM_GRAPHNOTIFY  WM_USER+13
#define HELPER_RELEASE(x) { if (x) x->Release(); x = NULL; }
```

3. Declare variables.

```
HWND      ghApp;
HINSTANCE ghInst;
HRESULT   hr;
LONG      evCode;
LONG      evParam1;
LONG      evParam2;
```

The *ghApp* variable is the handle of window to notify when the graph signals an event. The *ghInst* variable is the HINSTANCE of the window. The *evCode* variable will hold the event code, and the *evParam1* and *evParam2* variables will hold the event parameters.

4. Declare and initialize the necessary interfaces. The reference count of the interfaces is automatically incremented on initialization, so you don't need to call the IUnknown::AddRef method on them. For this example, you need only the four interfaces shown in the following code. For more information, see the documentation for the IMediaEventEx, IGraphBuilder, IMediaControl, and IVideoWindow interfaces.

```
IGraphBuilder *pigb  = NULL;
IMediaControl *pimc  = NULL;
IMediaEventEx *pimex = NULL;
IVideoWindow  *pivw  = NULL;
IFilterGraph  *pifg  = NULL;
IBaseFilter   *pifPP = NULL;
ISpecifyPropertyPages *pispp = NULL;
```

5. Define the function and declare variables. The *szFile* parameter is the name of the MPEG video file that will be played.

```
void MpegProp (LPSTR szFile)
{
```

6. Create a Unicode (wide character) string from the input file name.

```
WCHAR wFile[MAX_PATH];
MultiByteToWideChar( CP_ACP, 0, szFile, -1, wFile, MAX_PATH );
```

7. Instantiate the filter graph manager, asking for the IGraphBuilder interface.

```
hr = CoCreateInstance(CLSID_FilterGraph,
NULL,
CLSCTX_INPROC_SERVER,
IID_IGraphBuilder,
(void **)&pigb);
```

8. Query for the IMediaControl interface (provides the methods to run, pause, and stop the playback), the IMediaEventEx interface (so you can receive event notifications), and the IVideoWindow interface to hide the window when the movie is finished playing.

```
pigb->QueryInterface(IID_IMediaControl, (void **)&pimc);
pigb->QueryInterface(IID_IMediaEventEx, (void **)&pimex);
pigb->QueryInterface(IID_IVideoWindow, (void **)&pivw);
```

9. Ask the filter graph manager to build the filter graph that renders the input file. This does not play the media file. (When you play the file with **Run**, the filter graph will automatically render the input file's media type. You do not have to specify a renderer filter.)

```
hr = pigb->RenderFile(wFile, NULL);
```

10. Use a window to capture graph signal events. This not only improves performance, but allows your application to run in any threading model.

```
pimex->SetNotifyWindow((OAHWND)ghApp, WM_GRAPHNOTIFY, 0);
```

The window specified by *ghApp* will handle messages in response to all events from the graph. If an event occurs, DirectShow posts a WM_GRAPHNOTIFY message to the window.

11. Query for the IFilterGraph interface. Through **IFilterGraph**, you will retrieve a pointer to the IBaseFilter interface on the MPEG Video Codec filter. The easiest way to find the single MPEG video codec in the graph is through IFilterGraph::FindFilterByName.

```
pigb->QueryInterface(IID_IFilterGraph, (void **)&pifg);
```

12. Use FindFilterByName to find the MPEG Video Codec. This method returns a pointer (*&pifPP*) to the IBaseFilter interface on the MPEG Video Codec.

```
hr = pifg->FindFilterByName(L"MPEG Video Codec", &pifPP);
```

13. Retrieve the ISpecifyPropertyPages interface for the MPEG Video Codec. This filter has a property page that enables you to play MPEG files in color or monochrome.

```
    pif->QueryInterface(IID_ISpecifyPropertyPages, (void **)&pispp);
```

14. Allocate a counted array of GUIDs for the property page. The ISpecifyPropertyPages::GetPages method uses the COM **CAUUID** structure to receive an array of CLSIDs from the filter for each of its property pages. The structure has two members, **cElems**, which holds the number of property pages, and **pElems**, which points to an array holding the property page CLSIDs.

```
    CAUUID caGUID;
```

15. Using the pointer to the MPEG Video Decoder filter's property page, *pispp*, call the COM ISpecifyPropertyPages::GetPages method to fill the caGUID structure with a counted array of UUIDs, where each UUID specifies a property page CLSID.

```
    pispp->GetPages(&caGUID);
```

16. Release the ISpecifyPropertyPages interface.

```
   HELPER_RELEASE(pispp);
```

17. Create a modal dialog box to display the MPEG Video Decoder filter's property page. This dialog box appears before the video is played, enabling the user to choose to play back in color or monochrome.

```
    OleCreatePropertyFrame(NULL,
             0,
             0,
             L"Filter",          // Caption for the dialog box
             1,                  // Number of filters
             (IUnknown **)&pifPP,  // Pointer to the filter whose property
                                 // Pages are being displayed. This can
                                 // be an array of pointers if more than
                                 // one filter's property pages are to
                                 // be displayed. Note that only
                                 // properties common to all the filters
                                 // can be displayed on the same modal
                                 // dialog.
             caGUID.cElems,      // Number of property pages
             caGUID.pElems,      // Pointer to property page CLSIDs
             0,
             0,
             NULL);
```

18. Release the IBaseFilter interface.

```
   HELPER_RELEASE(pifPP);
```

The MPegProp sample uses the same WndMainProc callback function to handle the filter graph messages, the same GetClipFileName function to get the movie to be played, and the same the WinMain function to create the window as the Playfile sample. These are all generic windows functions.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

Use Multimedia Streaming in DirectShow Applications

This section describes and demonstrates how to support multimedia streaming in Microsoft® DirectShow™ applications. DirectShow applications typically use multimedia streaming to send audio and video data directly to a Microsoft DirectDraw® surface for rendering, instead of attaching playback to a specific window. This section has short conceptual explanations of windowless playback and multimedia streams, as well as additional detail on the multimedia streaming architecture and a minimal code demonstration of using streams to perform windowless playback of DirectShow-supported media files.

This section contains the following topics.

- Windowless Playback
- Multimedia Streams
- Code Walk-through of a Simple Application

Programmers who want to use multimedia streaming in their applications should be familiar with COM programming concepts, DirectDraw and its associated objects, and DirectShow media playback. For information on DirectDraw, consult the Microsoft DirectX SDK documentation. The DirectShow SDK documentation includes many examples of media file playback using C/C++; see About DirectShow and the included samples for more information. If you need information on programming with COM and OLE, consult reference materials such as *Inside OLE* by Kraig Brockschmidt or *Understanding ActiveX and OLE* by David Chappell.

Windowless Playback

Typically, applications display their video output in a clearly bounded rectangle, — the window. Each window has certain properties in common with other windows, such as menus, close buttons, and so forth. This shared behavior is helpful because it provides a measure of consistency and reliability to programming procedures and the user interface. DirectShow typically uses windows for media playback, because of the low programming overhead and consistent interface. However, there are a number of situations where an application developer wants to divorce media playback from the window and gain complete control over its appearance. For example, if you were creating a three-dimensional computer model of a museum tour, complete with moving exhibits and an animated tour guide, it would not be appropriate (or lifelike) to show each element of the tour in a separate window; you would need to integrate all of the elements together into a single presentation. By attaching the media playback to a DirectDraw surface instead of a window, you gain complete control over its appearance and behavior.

DirectDraw surfaces represent a portion of a system's video memory. Once you designate a surface as the destination of a movie's video data, you can blit the data to the surface in the same way you would normally blit color and texture information. Because it is a normal DirectDraw surface, you can manipulate it in any manner supported by the DirectDraw interfaces; you can play it back as the background of a game, texture map it into a three-dimensional environment, and so forth. While this level of control adds some programming overhead to your application, these effects would be impossible to do in a normal window.

Multimedia Streams

Audio and video data is, at its most basic, a sequence of information that specifies characteristics like color, resolution, frequency, and volume. Because there are a large number of devices and data formats related to media, moving data from its origin to its destination is a very convoluted process; you must know exactly how the original device formats its information, what characteristics the display format has, and how to convert the device information from its original format to a format suitable for rendering or storage. Because the exact steps in this process are different for every device, it is often difficult to handle multiple devices (such as a video camera, movie data file, and Internet URL) in a single application. Applications can, however, avoid much of this difficulty by using multimedia streaming as the data source. The streaming architecture automatically handles the process of data conversion and formatting, producing a consistently formatted data source ready for rendering or file storage. Thus, applications only need to handle the presentation of the data and not the data conversion.

Code Walk-through of a Simple Application

Using multimedia streams in a DirectShow application is fairly straightforward; the following steps describe the process.

1. Open a media file that DirectShow supports.
2. Create a multimedia stream for each of the file's media types; typically, this will be one video and one audio stream.
3. Create a DirectDraw surface and associate it with the video stream.
4. Render the stream data, which will then play back on the surface.

The following code sample, which you can find in its entirety in the \Streams\Simple\Main.cpp file included with the DirectShow SDK, demonstrates these steps. The complete file comprises three functions: OpenMMStream, RenderStreamToSurface, and main. OpenMMStream creates the audio and video multimedia streams from the media file, RenderStreamToSurface does the actual surface rendering, and main calls the other two functions appropriately. Because this example is a command-line application, you must supply the name of the media file as a parameter when you run the program. In Main.cpp, the following macro handles error checking.

```
#define CHECK_ERROR(x)        \
   if (FAILED(hr = (x))) { \
       printf(#x "  failed with HRESULT(0x%8.8X)\n", hr); \
       goto Exit;             \
   }
```

Each application that uses multimedia streaming must include the correct header files. The following list contains the stream-related header files from Main.cpp; the DirectShow SDK includes all of these header files.

```
#include "ddraw.h"       // DirectDraw interfaces
#include "mmstream.h"    // Multimedia stream interfaces
#include "amstream.h"    // DirectShow multimedia stream interfaces
#include "ddstream.h"    // DirectDraw multimedia stream interfaces
```

The code in Main.cpp is intended to be the minimum amount of programming necessary to implement multimedia streams, so it is appropriate to read it as a series of required steps. The following instructions illustrate all of the important concepts from Main.cpp, but don't necessarily include every line of code. For the complete code, refer to Main.cpp.

Creating a Multimedia Stream Linked to a DirectShow File

To create a multimedia stream and link it to a media file, perform the following steps. You do not necessarily need to perform the steps in the given order.

1) The OpenMMStream function creates a multimedia stream and attaches the stream to a valid input media file. The *pszFileName* parameter specifies the name of the media file, whose type DirectShow must support. The *pDD* parameter specifies an IDirectDraw interface that points to the destination DirectDraw object. When this function creates the multimedia stream, it attaches the stream's video portion to the object by using this pointer. The *ppMMStream* parameter represents a global stream pointer. Once this function creates a valid local stream, it points this parameter to the stream so other functions can use the stream as needed.

```
HRESULT OpenMMStream(const char * pszFileName, IDirectDraw *pDD,
                        IMultiMediaStream **ppMMStream) {
```

2) Declare a local IAMMultiMediaStream pointer, create a stream object, and initialize it. You should use the local *pAMStream* pointer during the stream's creation; don't use the global *ppMMStream* pointer until you are sure the stream and its media file are valid.

```
    *ppMMStream = NULL; // Initialize global stream pointer to null
    IAMMultiMediaStream *pAMStream;
    HRESULT hr;         // Function's return value

    CHECK_ERROR(CoCreateInstance(CLSID_AMMultiMediaStream, NULL,
                                  CLSCTX_INPROC_SERVER, IID_IAMMultiMedias
                                  (void **)&pAMStream));
    CHECK_ERROR(pAMStream->Initialize(STREAMTYPE_READ,
                                  AMMSF_NOGRAPHTHREAD, NULL));
```

3) Now that you have a stream object, add a single audio and video stream to it; typically, you need only these two streams for media file playback. When the IAMMultiMediaStream::AddMediaStream method receives the MSPID_PrimaryVideo flag as its second parameter, it uses the pointer in the first parameter as the destination surface for video playback. The audio stream needs no such surface, however, so you pass NULL as the first parameter when you add audio streams. The AMMSF_ADDDEFAULTRENDERER flag automatically adds the default sound renderer to the current filter graph.

```
    CHECK_ERROR(pAMStream->AddMediaStream(pDD, MSPID_PrimaryVideo, 0, NULL));
    CHECK_ERROR(pAMStream->AddMediaStream(NULL, MSPID_PrimaryAudio,
                                          AMMSF_ADDDEFAULTRENDERER, N
```

4) Convert the provided file name to a wide (Unicode) string and open the file. If the file name specifies a valid media file, DirectShow attaches the audio and video tracks to the streams you created earlier in the function. Point the *ppMMStream* parameter to the stream and increment the pointer's reference count.

```
    WCHAR       wPath[MAX_PATH];                // Wide (32-bit) string name
    MultiByteToWideChar(CP_ACP, 0, pszFileName, -1, wPath,
                                 sizeof(wPath)/sizeof(wPath[0]));

    CHECK_ERROR(pAMStream->OpenFile(wPath, 0));
    *ppMMStream = pAMStream;            // Set global pointer to local pointer
    pAMStream->AddRef();                // Add a reference to the file
```

Now that you have valid streams and a pointer to them, this function is complete.

Render the Video Data to a DirectDraw Surface

To render the video portion of a multimedia stream to a DirectDraw surface, perform the following steps. You do not necessarily need to perform the steps in the given order.

1) The RenderStreamToSurface function handles the actual rendering; it creates and initializes the required DirectDraw surface, and blits the video stream's data to the surface. The *pDD* parameter points to a global DirectDraw object, which you later use to create the surface. The *pPrimary* parameter is the primary rendering surface; it sends all blitted video data from the video stream, which the *pMMStream* parameter points to.

```
HRESULT RenderStreamToSurface(IDirectDraw *pDD, IDirectDrawSurface *pPrimary,
                              IMultiMediaStream *pMMStream) {
```

2) Create local variables for the surface, media streams, and video sample. When you blit data to the DirectDraw surface, you will use these local variables to store the individual frame and video sample information.

```
    IMediaStream *pPrimaryVidStream = NULL;
    IDirectDrawMediaStream *pDDStream = NULL;
    IDirectDrawSurface *pSurface = NULL;
    IDirectDrawStreamSample *pSample = NULL;
```

3) Retrieve the video stream from the global stream, which the *pMMStream* pointer specifies; the IMultiMediaStream::GetMediaStream method associates the local IMediaStream pointer with the retrieved stream. You can then use that pointer to obtain a DirectDraw media stream pointer, which you will need to retrieve the video format.

```
    CHECK_ERROR(pMMStream->GetMediaStream(MSPID_PrimaryVideo,
                    &pPrimaryVidStream));
    CHECK_ERROR(pPrimaryVidStream->QueryInterface(
                    IID_IDirectDrawMediaStream, (void **)&pDDStream));
```

4) Create a DirectDraw surface and a bounding rectangle to use for playback. Call IDirectDrawMediaStream::GetFormat to retrieve the video format and set the dimensions of the rectangle to match the format dimensions.

```
    DDSURFACEDESC ddsd;                    // Surface characteristics
    ddsd.dwSize = sizeof(ddsd);

    CHECK_ERROR(pDDStream->GetFormat(&ddsd, NULL, NULL));
    RECT rect;                             // Playback rectangle
    rect.top = rect.left = 0;
    rect.bottom = ddsd.dwHeight;
    rect.right = ddsd.dwWidth;

    CHECK_ERROR(pDD->CreateSurface(&ddsd, &pSurface, NULL));
```

5) Create the first video sample and attach it to the desired playback surface. You can then blit all samples from the video stream directly to the surface by calling the DIrectDraw Surface's Update method in a loop. Each loop iteration throws out the previous video image and grabs the next image from the stream. The loop breaks once there is no remaining renderable video data.

```
    CHECK_ERROR(pDDStream->CreateSample(pSurface, NULL, 0, &pSample));

    while (true) {
        if (pSample->Update(0, NULL, NULL, 0) != S_OK) {
            break;
        }
        pPrimary->Blt(&rect, pSurface, &rect, DDBLT_WAIT, NULL);
    }
```

6) Release all local pointers.

```
    RELEASE(pPrimaryVidStream);
    RELEASE(pDDStream);
    RELEASE(pSample);
    RELEASE(pSurface);

    return hr;
}
```

Once DirectShow finishes rendering all available data, the function is complete.

Run the Program

To obtain a valid media filename and run the program, perform the following steps. You do not necessarily need to perform the steps in the given order.

1) Create a main function to obtain the file name and run the rendering process. The following example takes the media file name as a command-line parameter.

```
int main(int argc, char *argv[]) {
```

2) Create a global DirectDraw object; once you have a valid object, create a surface that you will later use for video playback. This example calls the Win32 **GetDesktopWindow** function to associate the surface with the desktop, reducing the amount of required configuration code.

```
    CoInitialize(NULL);         // Initialize the COM objects

    // Create the DirectDraw object and its interface pointer
    IDirectDraw *pDD;
    HRESULT hr = DirectDrawCreate(NULL, &pDD, NULL);

    if (SUCCEEDED(hr)) {        // The object is valid
        DDSURFACEDESC ddsd;     // Surface characteristics
        IDirectDrawSurface *pPrimarySurface;

        pDD->SetCooperativeLevel(GetDesktopWindow(), DDSCL_NORMAL);
        ddsd.dwSize = sizeof(ddsd);
        ddsd.dwFlags = DDSD_CAPS;
        ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE;
        hr = pDD->CreateSurface(&ddsd, &pPrimarySurface, NULL);
```

3) Create the multimedia stream and call the previously defined functions. Once the functions finish execution, make sure to release all pointers at the correct times. Once playback is complete, call the Win32 **CoUninitialize** function and return. Once DirectShow finishes playback of the file, it returns control to the command line.

```
        if (SUCCEEDED(hr)) {
            IMultiMediaStream *pMMStream;
            hr = OpenMMStream(argv[1], pDD, &pMMStream);
            if (SUCCEEDED(hr)) {
                RenderStreamToSurface(pDD, pPrimarySurface, pMMStream);
                pMMStream->Release();
            }
            pPrimarySurface->Release();
    }
    CoUninitialize();   // Release COM objects
    return 0;           // Success
}
```

Now that you know how to direct streamed video data to a DirectDraw surface, you can use this functionality any way you would normally use DirectDraw surfaces. A typical use would be to texture map the playback surface onto a Direct3D primitive object and incorporate it as part of a three-dimensional environment. For information on controlling any part of DirectDraw, consult the DirectX SDK documentation.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

Play a Movie in a Window Using DirectDrawEx and Multimedia Streaming

This article walks through the MovieWin C++ example code, which plays movies in a window by rendering to a Microsoft DirectDraw® surface. The MovieWin example code is a Microsoft®

Windows® 95 application that is an extension of the ShowStrm Sample (Multimedia Streaming Application) sample. MovieWin uses multimedia streaming to render a video file to a DirectDraw surface created through DirectDrawEx. It implements a primary DirectDraw surface and an offscreen DirectDraw surface to optimize frame blitting. It also attaches a DirectDraw clipper to the window to process window overlapping.

Contents of this article:

The example demonstrates a way to render a movie that differs from the traditional method of instantiating a filter graph directly in your application. The MovieWin example code uses the multimedia streaming interfaces to automatically negotiate the transfer and conversion of data from the source to the application, so you don't have to write code to handle the connection, transfer of data, data conversion, or actual data rendering.

Additionally, the example demonstrates how to create DirectDraw surfaces and how to add code for a DirectDrawClipper object through DirectDrawEx.

Note that all error checking has been left out of the code walk-through. The Entire MovieWin Example Code section provides all of the code with complete error checking.

Necessary Header files and Libraries

This section discusses necessary headers and libraries that need to be included and examines each function in the MovieWin example code in detail.

To compile the MovieWin example code you must have DirectX Media SDK 5.*x* or later installed and you will need to set your include path under Tools/Options/Directories/Include to c:\DXMedia\Include and your library path to c:\DXMedia\Lib. Also link with the Amstrmid.lib, the Quartz.lib, the Strmbase.lib, and the Ddraw.lib (DirectDrawEx does not provide its own library) libraries under Project/Settings/Link.

Include the necessary header files and define the window's name and the window class name.

```
#include <windows.h>
#include <mmstream.h>   // Multimedia stream interfaces
#include <amstream.h>   // DirectShow multimedia stream interfaces
#include <ddstream.h>   // DirectDraw multimedia stream interfaces
#include <initguid.h>   // Defines DEFINE_GUID macro and enables GUID initializatic
#include <ddrawex.h>    // DirectDrawEx interfaces
#include "resource.h"   // Resources for the menu bar

#define APPLICATIONNAME "Multimedia Stream In Window"
```

```
#define CLASSNAME "MMSDDRAWEXWINDOW"
```

Then declare the following global variables:

```
HWND              ghWnd;
HINSTANCE         ghInst;
BOOL              g_bAppactive=FALSE,    // The window is active
                  g_bFileLoaded = FALSE, // There is a file loaded
                  g_bPaused=FALSE;       // The movie has been paused
RECT              rect, rect2;          // Rectangles for screen coordinates
```

The ghWnd variable is the handle of the window to send messages to. The ghInst variable is the handle of the instance of the window. The three Boolean values g_bAppactive, g_bFileLoaded, g_bPaused variables are used to determine the various states of the application and are used extensively by the WndMainProc function. They are declared as global variables to retain their TRUE or FALSE status. Finally, rect and rect2 are rectangle structures that will contain the original movie coordinates and the coordinates of the window to show the movie in, respectively.

Next, declare the DirectDrawEx and multimedia streaming interfaces. The reference count of the interfaces is automatically incremented on initialization, so you don't need to call the IUnknown::AddRef method to increment them. For more information on these interfaces, see DirectDrawEx, , and the Microsoft DirectX® SDK.

```
//DirectDrawEx Global interfaces
IDirectDraw            *g_pDD=NULL;
IDirectDraw3           *g_pDD3=NULL;
IDirectDrawFactory     *g_pDDF=NULL;
IDirectDrawSurface     *g_pPrimarySurface=NULL,
                       *g_pDDSOffscreen=NULL;
IDirectDrawClipper     *g_pDDClipper=NULL;

//Global MultiMedia streaming interfaces
IMultiMediaStream        *g_pMMStream=NULL;
IMediaStream             *g_pPrimaryVidStream=NULL;
IDirectDrawMediaStream   *g_pDDStream=NULL;
IDirectDrawStreamSample  *g_pSample=NULL;
```

Finally, declare the function prototypes.

```
//Function prototypes
int PASCAL WinMain(HINSTANCE hInstC, HINSTANCE hInstP, LPSTR lpCmdLine, int nCmdShc
LRESULT CALLBACK WndMainProc (HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam
HRESULT InitDDrawEx();
BOOL GetOpenMovieFile(LPSTR szName);
HRESULT RenderFileToMMStream(LPCTSTR szFilename);
HRESULT InitRenderToSurface();
void RenderToSurface();
void ExitCode();
```

WinMain Function

The WinMain function is a generic Windows function with a few exceptions.

Immediately after the Win32 CreateWindowEx function, the InitDDrawEx function is called to initialize the DirectDrawEx surfaces that the movie will play on and to create a clipper to attach to the window. The clipper can only be created after it has a global handle to the window (ghWnd), and so must be created after the call to the **CreateWindowEx** function has returned.

The message pump is a standard Windows message pump containing the TranslateMessage and the DispatchMessage functions with an interesting note. Before the code reaches these functions, it calls the PeekMessage function. The **PeekMessage** function checks a thread message queue for a message and places the message (if any) in the specified structure. If there are messages being passed to the window the code proceeds to the regular GetMessage, TranslateMessage, and DispatchMessage functions respectively. However if there are no messages in the message queue, the process will check for the g_bFileLoaded Boolean value, which specifies whether a file has been loaded. Initially, the value in g_bFileLoaded is FALSE so the code maintains its loop, waiting for new messages.

After a file has been loaded and rendered to a multimedia stream (see GetOpenMovieFile function and RenderFileToMMStream function) the g_bFileLoaded value and the g_bAppactive values are set to TRUE and the message pump will call the RenderToSurface function, which blits one frame of the movie to the window's coordinates. As the loop continues, the movie continues to render frame by frame until completion or until it is interrupted the PeekMessage function with an outside message to the window. If the movie is paused, stopped, or if it completes on its own, the g_bAppactive variable is set to FALSE, which causes the call to RenderToSurface to be skipped until g_bAppactive is set to TRUE again.

The following code shows how to create the message pump.

```
while(1){
                //The PeekMessage function checks a thread message queue
                //for a message and places the message (if any) in the specified s
                if(PeekMessage(&msg, NULL, 0,0,PM_NOREMOVE)){

                        // Quit if WM_QUIT found
                        if(!GetMessage(&msg,NULL, 0, 0)) return (msg.wParam);

                        // Otherwise handle the messages
                             TranslateMessage(&msg);         // Allow input
                             DispatchMessage(&msg);          // Send to appropri
                }
                else{
                        // If there are no other windows messages...
                        // Render frame by frame (but only if the App is the activ
                        // window and a file is actually loaded)
                        if (g_bFileLoaded && g_bAppactive) {
                             RenderToSurface();
                             }
                        }
                }
          return msg.wParam;
```

Initialize DirectDraw Surfaces and Create the Clipper

The InitDDrawEx function initializes a primary DirectDraw surface and an offscreen DirectDraw surface, as well as a clipper object that is attached to the window. The following code shows how to do this.

1. Declare local variables and initialize the COM subsystem.

```
HRESULT               hr=NOERROR;
DDSURFACEDESC   ddsd, ddsd2;

CoInitialize(NULL);
```

2. Create the DirectDrawFactory object and expose the IDirectDrawFactory interface.

```
CoCreateInstance(CLSID_DirectDrawFactory, NULL, CLSCTX_INPROC_SERVER,
                                                          IID_IDirectDrawFactory
```

   Use the pointer to the IDirectDrawFactory interface to call the IDirectDrawFactory::CreateDirectDraw method, which you use to create the DirectDraw object, set the cooperative level, and get the address of an IDirectDraw interface pointer.

```
g_pDDF->CreateDirectDraw(NULL, GetDesktopWindow(), DDSCL_NORMAL,
                                    NULL, NULL, &g_pDD);
```

3. Query for the IDirectDraw3 interface, which you use to create the DirectDraw surfaces.

```
g_pDD->QueryInterface(IID_IDirectDraw3, (LPVOID*)&g_pDD3);
```

4. Initialize the DDSURFACEDESC structure for the primary surface. The following is the minimum code needed to accomplish this. You should also initialize other members of the structure here if your code must create more sophisticated applications.

```
ZeroMemory(&ddsd, sizeof(ddsd));
ddsd.dwSize = sizeof(ddsd);
ddsd.dwFlags = DDSD_CAPS;
ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE;
```

5. Call the IDirectDraw3::CreateSurface method to create the primary DirectDraw surface and return a pointer to IDirectDrawSurface interface.

```
g_pDD3->CreateSurface(&ddsd, &g_pPrimarySurface, NULL);
```

6. Create the offscreen surface where the IStreamSample::Update method will send the individual movie frames before they are blitted onto the screen. Using an offscreen surface optimizes the performance of the video and enables the blits to be processed at a faster rate. Also the video remains in memory and can be called upon in the event of a repaint notification.

   You must create the offscreen surface with the identical height, width, and pixel format to the primary surface in order to blit from one to the other. Do this by first getting the DDSURFACEDESC structure from the primary surface through a call to the IDirectDrawSurface::GetSurfaceDesc method.

```
g_pPrimarySurface->GetSurfaceDesc(&ddsd);
```

7. Now you can initialize the DDSURFACEDESC structure for the offscreen surface with the same parameters as the primary surface:

```
ZeroMemory(&ddsd2, sizeof(ddsd2));
ddsd2.dwSize = sizeof(ddsd2);
ddsd2.dwFlags = DDSD_CAPS | DDSD_HEIGHT | DDSD_WIDTH | DDSD_PIXELFORMAT;
ddsd2.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN;
ddsd2.dwHeight = ddsd.dwHeight; //set the height of the surfaces equal
ddsd2.dwWidth  = ddsd.dwWidth;  //set the width of the surfaces equal
ddsd2.ddpfPixelFormat = ddsd.ddpfPixelFormat; //set the pixel formats equal
```

8. Call the IDirectDraw3::CreateSurface method to create the offscreen surface.

```
g_pDD3->CreateSurface(&ddsd2, &g_pDDSOffscreen, NULL)
```

   At this point, you should have two identical DirectDraw surfaces: the offscreen surface that will be used to update the movie frames on, and the primary surface, which your user will see. The primary surface will contain the video after the data has been blitted from the offscreen surface to the primary surface.

9. To give the window the look and feel of a regular window, you must add code for a clipper. The DirectDrawClipper object (casually referred to as a "clipper") helps you prevent blitting to certain portions of a surface or beyond the bounds of a surface. DirectDrawClipper objects expose their functionality through the IDirectDrawClipper interface. You can create a clipper by calling the IDirectDraw3::CreateClipper method.

   Use the following code to create the clipper object and retrieve a pointer to the IDirectDrawClipper interface.

```
g_pDD3->CreateClipper(0, &g_pDDClipper, NULL);
```

10. Use the IDirectDrawSurface interface to attach the clipper to the primary surface.

```
g_pPrimarySurface->SetClipper(g_pDDClipper);
```

11. Finally, associate the clipper with the window by calling the IDirectDrawClipper::SetHWnd method.

```
g_pDDClipper->SetHWnd(0, ghWnd);
```

At this point, you should have two DirectDraw surfaces, and a clipper attached to the primary surface and to the applications window. The DirectDrawEx initialization is complete and all the objects are available to the process until the ExitCode function is called to release the objects.

For more information on DirectDrawEx, see DirectDrawEx.

Open a Movie File

The following code shows how to use the GetOpenMovieFile function to display the Open file dialog box. It initializes the OPENFILENAME structure and calls the GetOpenFileName API.

```
BOOL GetOpenMovieFile(LPSTR szName)
{
        OPENFILENAME    ofn;

        ofn.lStructSize        = sizeof(OPENFILENAME);
        ofn.hwndOwner          = ghWnd;
        ofn.lpstrFilter        = NULL;
        ofn.lpstrFilter        = "Video (*.avi;*.mpg;*.mpeg)\0*.avi;*.mpg;*.mpeg\0A
        ofn.lpstrCustomFilter = NULL;
        ofn.nFilterIndex      = 1;
        *szName = 0;
        ofn.lpstrFile         = szName;
        ofn.nMaxFile          = MAX_PATH;
        ofn.lpstrInitialDir   = NULL;
        ofn.lpstrTitle        = NULL;
        ofn.lpstrFileTitle    = NULL;
        ofn.lpstrDefExt       = NULL;
        ofn.Flags             = OFN_FILEMUSTEXIST | OFN_READONLY | OFN_PATHMUSTEXI:
        return GetOpenFileName((LPOPENFILENAME)&ofn);
}
```

Create the Multimedia Stream Object

The RenderFileToMMStream function creates a multimedia stream and attaches the stream to the file retrieved by the GetOpenMovieFile function. This function uses the IAMMultiMediaStream interface to expose DirectShow functionality to the application. After the address of a pointer to the **IAMMultiMediaStream** interface is retrieved, it will be used to initialize the stream, add specific media streams to the current filter graph, and open and automatically create a filter graph for the specified media file.

The following steps show how to do this.

1. Declare the local variables hr and pAMStream, and convert the provided file name to a wide (Unicode) string.

```
HRESULT hr;
IAMMultiMediaStream *pAMStream=NULL;
WCHAR wFile[MAX_PATH];
MultiByteToWideChar(CP_ACP, 0, szFilename, -1, wFile,
                        sizeof(wFile)/sizeof(wFile[0]));
```

2. Create the AMMultiMediaStream object and initialize it.

```
hr =CoCreateInstance(CLSID_AMMultiMediaStream, NULL, CLSCTX_INPROC_SERVER,
                                    IID_IAMMultiMediaStream, (void **)&pAMSt
hr = pAMStream->Initialize(STREAMTYPE_READ, 0, NULL);
```

3. Now that you have a stream object, add a single audio and video stream to it; typically, you need only these two streams for media file playback. When the IAMMultiMediaStream::AddMediaStream method receives the MSPID_PrimaryVideo flag as its second parameter, it uses the pointer in the first parameter as the destination surface for video playback. The audio stream needs no such surface, however, so pass NULL as the first parameter when you add audio streams. The AMMSF_ADDDEFAULTRENDERER flag automatically adds the default sound renderer to the current filter graph.

```
hr = pAMStream->AddMediaStream(g_pDD3, &MSPID_PrimaryVideo, 0, NULL);
hr = pAMStream->AddMediaStream(NULL, &MSPID_PrimaryAudio, AMMSF_ADDDEFAULTRENI
```

4. Finally, open and create a filter graph for the specified media file and save the local stream to the global variable g_pMMStream. Don't forget to increase the reference count on the IAMMultiMediaStream object.

```
//Opens and automatically creates a filter graph for the specified media file
```

```
hr = pAMStream->OpenFile(wFile, 0);
//save the local stream to the global variable
g_pMMStream = pAMStream;
// Add a reference to the file
pAMStream->AddRef();
```

Now that you have valid streams and a pointer to them, this function is complete. For more information on multimedia streams see and Use Multimedia Streaming in DirectShow Applications.

Create the Stream Sample Object

The InitRenderToSurface function creates the stream sample that will be associated with the offscreen DirectDrawSurface object. The stream sample will be used later by the RenderToSurface function to call the IStreamSample::Update method to perform frame-by-frame updates of the sample.

The following steps show how to do this.

1. To create and initialize the stream sample, declare the local variables, and then get the primary video media stream by using the IMultiMediaStream::GetMediaStream method.

   ```
   HRESULT                  hr;
   DDSURFACEDESC    ddsd;

   //Use the multimedia stream to get the primary video media stream
   hr = g_pMMStream->GetMediaStream(MSPID_PrimaryVideo, &g_pPrimaryVidStream);
   ```

2. After you obtain the primary video stream interface (IMediaStream), you can use it to query for the IDirectDrawMediaStream interface, which you'll use to create the stream sample.

   ```
   hr = g_pPrimaryVidStream->QueryInterface(IID_IDirectDrawMediaStream, (void **)
   ```

3. Before you can create the stream sample, you must call the IDirectDrawMediaStream::GetFormat method. The trick to watch on this call is that you must set the **dwSize** member of the DDSURFACEDESC structure. After the stream sample has retrieved the height and width of the movie file, you can set the rectangle that the offscreen surface will use to contain the video data.

   ```
   ddsd.dwSize = sizeof(ddsd);
   hr = g_pDDStream->GetFormat(&ddsd, NULL, NULL, NULL);
   rect.top = rect.left = 0;
   rect.bottom = ddsd.dwHeight;
   rect.right = ddsd.dwWidth;
   ```

4. Create the stream sample by calling the IDirectDrawMediaStream::CreateSample method with the offscreen surface and the RECT structure, which was just initialized with the movie coordinates. This method will retrieve a pointer to the global IDirectDrawStreamSample interface *g_pSample*.

   ```
   hr = g_pDDStream->CreateSample(g_pDDSOffscreen, &rect, 0, &g_pSample);
   ```

At this point, the IDirectDrawMediaStream::CreateSample method has created a global IDirectDrawStreamSample stream sample and returned a pointer to *g_pSample*, its interface, which the RenderToSurface function can use.

Render the Multimedia Stream to the DirectDraw Surface

The RenderToSurface function handles the actual rendering and blits the video stream's data to the primary surface. The main message pump in the WinMain function calls this method. The RenderToSurface function performs one individual frame update at a time and one blit from the offscreen surface to the primary surface. When the movie is complete, it will set the stream state to STOP.

The following steps show how to do this.

1. Declare the local variables.

    ```
    HRESULT         hr;
    POINT           point;
    ```

2. Call the IStreamSample::Update method. Each loop iteration throws out the previous video image and grabs the next image from the stream.

    If the update is successful, the Microsoft Win32® GetClientRect and the ClientToScreen functions are called to get the rectangle coordinates of the window into which the video will be displayed. These functions must be called after each update, in case a user has moved or resized the window.

3. After the window's coordinates have been retrieved, call the IDirectDrawSurface3::Blt method to perform a bit block transfer of the movie's video data from the offscreen surface to the primary surface. The loop breaks and the stream state is set to STOP when no renderable video data remains.

    ```
    if (g_pSample->Update(0, NULL, NULL, 0) != S_OK) {
                    g_bAppactive = FALSE;
                    g_pMMStream->SetState(STREAMSTATE_STOP);
            }
            else {
            //get window coordinates to blit into
            GetClientRect(ghWnd, &rect2);
            point.x = rect2.top;
            point.y = rect2.left;
            ClientToScreen(ghWnd, &point);
            rect2.left = point.x;
            rect2.top = point.y;
            point.x = rect2.right;
            point.y = rect2.bottom;
            ClientToScreen(ghWnd, &point);
            rect2.right = point.x;
            rect2.bottom= point.y;

    //Blit from the offscreen surface to the primary surface
            hr = g_pPrimarySurface->Blt(&rect2, g_pDDSOffscreen, &rect, DDBLT_WAI
    ```

    This function will be called repeatedly from the WinMain function's message pump as long as the g_bAppactive and g_bFileLoaded Boolean values are TRUE.

Release Objects

The ExitCode function releases all objects that the MovieWin application creates, destroys the window, and closes the COM library.

Call this function if the application fails or the user quits the program.

```
void ExitCode()
{
        //Release MultiMedia streaming Objects
        if (g_pMMStream != NULL) {
                g_pMMStream->Release();
                g_pMMStream= NULL;
        }
        if (g_pSample != NULL) {
                g_pSample->Release();
                g_pSample = NULL;
        }
        if (g_pDDStream != NULL) {
                g_pDDStream->Release();
                g_pDDStream= NULL;
        }
        if (g_pPrimaryVidStream != NULL) {
                g_pPrimaryVidStream->Release();
                g_pPrimaryVidStream= NULL;
        }
        //Release DirectDraw Objects
        if (g_pDDF !=NULL) {
                g_pDDF->Release();
                g_pDDF = NULL;
        }
        if (g_pPrimarySurface!=NULL) {
                g_pPrimarySurface->Release();
                g_pPrimarySurface=NULL;
        }
        if (g_pDDSOffscreen !=NULL) {
                g_pDDSOffscreen->Release();
                g_pDDSOffscreen= NULL;
        }
        if (g_pDDClipper !=NULL) {
                g_pDDClipper->Release();
                g_pDDClipper=NULL;
        }
        if (g_pDD3 != NULL) {
                g_pDD3->Release();
                g_pDD3 = NULL;
        }
        if (g_pDD != NULL) {
                g_pDD->Release();
                g_pDD = NULL;
        }

        PostQuitMessage(0);
        CoUninitialize();
}
```

WndMainProc Function

The WndMainProc callback function handles any messages sent to the window and calls the ExitCode function when the user quits the application. Users generate messages by selecting various items from the menu, including Open, Start, Stop, Pause, About, and Exit.

If the user chooses Open, an IDM_OPEN message is generated and the following code runs.

```
//If a file is already open - call STOP first
```

```
if (g_bAppactive && g_bFileLoaded) {
        g_pMMStream->SetState(STREAMSTATE_S
}

bOpen = GetOpenMovieFile(szFilename);
if (bOpen) {
        hr = RenderFileToMMStream(szFilenam
        hr = InitRenderToSurface();
        g_bAppactive = g_bFileLoaded = TRUI
        g_bPaused = FALSE;              //T
        //Now set the multimedia stream to
        hr = g_pMMStream->SetState(STREAMST
}
break;
```

This code first checks whether a file is loaded (g_bFileLoaded) and if it is in a running state (g_bAppactive). If this is the case, the IMultiMediaStream::SetState method is called to stop the stream before another one is loaded through a call to the GetOpenMovieFile function. After the call to GetOpenMovieFile has returned successfully, the RenderFileToMMStream function is called, followed by the InitRenderToSurface function. If both of these functions are successful, the g_bFileLoaded and g_bAppactive Boolean values are set to TRUE and g_bPaused is set to FALSE in case the old file was in a paused state. Finally, the **IMultiMediaStream::SetState** method is called to set the state to RUN and now the RenderToSurface function will automatically be called through the WinMain function's message pump.

If the user chooses Play from the application's menu, an IDM_START message is generated and the following code runs.

```
if (g_bAppactive && g_bFileLoaded)
                                        {break;                         //
                                        }
                                        else {
                                                if (g_bPaused) {        // If its i
                                                        g_pMMStream->Seek(StreamTim
                                                        g_pMMStream->SetState(STREA
                                                        g_bAppactive = TRUE;
                                                        g_bPaused = FALSE;
                                                        }
                                                else {

                                        if (g_bFileLoaded) {    // If a file is act
                                                g_bAppactive = g_bFileLoaded = TRUI
                                                hr = g_pMMStream->SetState(STREAMST
                                        }
                                        else {
                                                MessageBox(hWnd, "Please select a m
                                        }
                                        }
                                        }
                                        break;
```

This code first checks if a file is loaded (g_bFileLoaded) and if it is in a running state (g_bAppactive). If this is the case, **break** is called to ignore the message. If the movie is in a paused state, the IMultiMediaStream::Seek method is called to seek to the correct location in the file, and then the IMultiMediaStream::SetState method is called to set the state to RUN again. The Boolean values g_bAppactive and g_bPaused are reset again to TRUE and FALSE respectively.

If a file is loaded but not in a paused state, it must be in a stopped state. Therefore, if this code succeeds on the if (g_bFileLoaded) call it must restart the movie from the beginning. This involves resetting the g_bAppactive Boolean value to TRUE and calling the IMultiMediaStream::SetState method to set the stream state to RUN.

If the user chooses Pause from MovieWin's menu, an IDM_PAUSE message is generated and the following code runs.

```
// Pause if not already in a paused state and you have a file loaded
                                             if (!g_bPaused &&g_bFileLoaded) {
                                                      hr = g_pMMStream->GetTime(&StreamTi
                                                      hr = g_pMMStream->SetState(STREAMST
                                                      g_bAppactive = FALSE;
                                                      g_bPaused       = TRUE;
                                             }
                                             break;                          // If its a
```

In order for the pause key to do anything, the application must not already be in a paused stated (!g_bPaused) and a file must be loaded (g_bFileLoaded). If these two conditions are both TRUE, the IMultiMediaStream::GetTime method is called to store the STREAM_TIME at which the application was paused in the static StreamTime variable, and then the IMultiMediaStream::SetState method set the stream state to STOP. Finally, the g_bAppactive and the g_bPaused global Boolean values must be set to FALSE and TRUE respectively.

If the user chooses Stop from the application's menu, an IDM_STOP message is generated and the following code executes.

```
if (g_bFileLoaded) {
                                                      g_pMMStream->SetState(STREAMSTATE_S
                                                      StreamTime = 0; // Reset the strean
                                                      g_pMMStream->Seek(StreamTime);  //R
                                                      g_pMMStream->SetState(STREAMSTATE_R
                                                      RenderToSurface();
                                                      g_pMMStream->SetState(STREAMSTATE_S
                                                      StreamTime = 0;
                                             }
                                             g_bAppactive = FALSE;
```

The preceding code runs if there is a file loaded (g_bFileLoaded). In such a case the IMultiMediaStream::SetState method sets the stream state to STOP and the global STREAM_TIME value is set to zero. Next, the IMultiMediaStream::Seek method and the **IMultiMediaStream::SetState** method are called to run one frame of the video before the true stop is called. After the RenderToSurface function renders the frame, the **IMultiMediaStream::SetState** method is called a final time to stop the video. This gives the user the visual experience of seeing the movie rewind to the beginning.

Finally, if the user chooses **Exit** from the application's menu, an IDM_EXIT message is generated and the following code runs.

```
response = MessageBox(hWnd, "Quit the Program?", "Quit", MB_YESNO);
                                             if (response==IDYES) SendMessage(ghWnd, WM_
                                             break;
```

When it runs, this code will prompt the user if he or she really wants to quit the application. If the user chooses Yes, a WM_DESTROY message is sent, which calls the ExitCode function.

Entire MovieWin Example Code

This is the entire code for the MovieWin example code. To compile this code in Microsoft Visual

Studio™, create a new Win32 application project and add this code into the project. Follow the directions in the following code comments on how to set your project libraries and include paths.

```
// This application uses a Multimedia stream to render
// a video file to a DirectDrawEx surface contained in
// a window. It implements a primary DirectDraw surface
// and an offscreen DirectDraw surface to optimize individual
// frame blits. It also attaches a DirectDraw clipper to the
// window to process window overlapping.


//To compile this program you must have DXMedia SDK 5.1 installed
//and you will need set your include path under tools/options/directories/include
//to c:\DXMedia\include and your library path to c:\DXMedia\lib
//Also link with the following libraries under project/settings/link...
//amstrmid.lib quartz.lib strmbase.lib ddraw.lib

#include <windows.h>
#include <mmstream.h>   // Multimedia stream interfaces
#include <amstream.h>   // DirectShow multimedia stream interfaces
#include <ddstream.h>   // DirectDraw multimedia stream interfaces
#include <initguid.h>   // Defines DEFINE_GUID macro and enables GUID initializatic
#include <ddrawex.h>    // DirectDrawEx interfaces
#include "resource.h"   // Resources for the menu bar

#define APPLICATIONNAME "Multimedia Stream In Window"
#define CLASSNAME "MMSDDRAWEXWINDOW"


//Global variables
HWND                     ghWnd;
HINSTANCE               ghInst;
BOOL                     g_bAppactive=FALSE,           // The window is active
                                   g_bFileLoaded = FALSE,  // There is a file loac
                                   g_bPaused=FALSE;            // The movie has be
RECT                     rect, rect2;                  // Rectangles for screen cc

//DirectDrawEx Global interfaces
IDirectDraw                    *g_pDD=NULL;
IDirectDraw3            *g_pDD3=NULL;
IDirectDrawFactory      *g_pDDF=NULL;
IDirectDrawSurface      *g_pPrimarySurface=NULL,
                                       *g_pDDSOffscreen=NULL;
IDirectDrawClipper      *g_pDDClipper=NULL;

//Global MultiMedia streaming interfaces
IMultiMediaStream              *g_pMMStream=NULL;
IMediaStream                   *g_pPrimaryVidStream=NULL;
IDirectDrawMediaStream  *g_pDDStream=NULL;
IDirectDrawStreamSample *g_pSample=NULL;

//Function prototypes
int PASCAL WinMain(HINSTANCE hInstC, HINSTANCE hInstP, LPSTR lpCmdLine, int nCmdShc
LRESULT CALLBACK WndMainProc (HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam
HRESULT InitDDrawEx();
BOOL GetOpenMovieFile(LPSTR szName);
HRESULT RenderFileToMMStream(LPCTSTR szFilename);
HRESULT InitRenderToSurface();
void RenderToSurface();
void ExitCode();
void ExitCode()
{
        //Release MultiMedia streaming Objects
        if (g_pMMStream != NULL) {
                g_pMMStream->Release();
                g_pMMStream= NULL;
        }
        if (g_pSample != NULL) {
                g_pSample->Release();
                g_pSample = NULL;
        }
        if (g_pDDStream != NULL) {
                g_pDDStream->Release();
                g_pDDStream= NULL;
        }
        if (g_pPrimaryVidStream != NULL) {
                g_pPrimaryVidStream->Release();
                g_pPrimaryVidStream= NULL;
        }
        //Release DirectDraw Objects
        if (g_pDDF !=NULL) {
                g_pDDF->Release();
                g_pDDF = NULL;
        }
        if (g_pPrimarySurface!=NULL) {
                g_pPrimarySurface->Release();
                g_pPrimarySurface=NULL;
        }
        if (g_pDDSOffscreen !=NULL) {
                g_pDDSOffscreen->Release();
                g_pDDSOffscreen= NULL;
        }
        if (g_pDDClipper !=NULL) {
                g_pDDClipper->Release();
                g_pDDClipper=NULL;
        }
        if (g_pDD3 != NULL) {
                g_pDD3->Release();
                g_pDD3 = NULL;
        }
        if (g_pDD != NULL) {
                g_pDD->Release();
                g_pDD = NULL;
        }

        PostQuitMessage(0);
        CoUninitialize();
}

//Create the stream sample which will be used to call updates on the video
HRESULT InitRenderToSurface()
{
        HRESULT                 hr;
        DDSURFACEDESC   ddsd;

        //Use the multimedia stream to get the primary video media stream
    hr = g_pMMStream->GetMediaStream(MSPID_PrimaryVideo, &g_pPrimaryVidStream);
        if (FAILED(hr))
        {   goto Exit;
        }
        //Use the media stream to get the IDirectDrawMediaStream
    hr = g_pPrimaryVidStream->QueryInterface(IID_IDirectDrawMediaStream, (void **)&
        if (FAILED(hr))
        {   goto Exit;
        }
        //Must set dwSize before calling GetFormat
    ddsd.dwSize = sizeof(ddsd);
    hr = g_pDDStream->GetFormat(&ddsd, NULL, NULL, NULL);
        if (FAILED(hr))
        {   goto Exit;
        }

          rect.top = rect.left = 0;
    rect.bottom = ddsd.dwHeight;
    rect.right = ddsd.dwWidth;

        //Create the stream sample
        hr = g_pDDStream->CreateSample(g_pDDSOffscreen, &rect, 0, &g_pSample);
        if (FAILED(hr))
        {   goto Exit;
        }
Exit:
        if (FAILED(hr))
        {       MessageBox(ghWnd, "Initialization failure in InitRenderToSurface",
                return E_FAIL;
        }
        return NOERROR;
}

//Perform frame by frame updates and blits. Set the stream
//state to STOP if there are no more frames to update.
void RenderToSurface()
{
        HRESULT         hr;
        POINT           point;

        //update each frame
        if (g_pSample->Update(0, NULL, NULL, 0) != S_OK) {
                g_bAppactive = FALSE;
                g_pMMStream->SetState(STREAMSTATE_STOP);
        }
        else {
        //get window coordinates to blit into
        GetClientRect(ghWnd, &rect2);
        point.x = rect2.top;
        point.y = rect2.left;
        ClientToScreen(ghWnd, &point);
        rect2.left = point.x;
        rect2.top = point.y;
        point.x = rect2.right;
        point.y = rect2.bottom;
        ClientToScreen(ghWnd, &point);
        rect2.right = point.x;
        rect2.bottom= point.y;

        //blit from the offscreen surface to the primary surface
        hr = g_pPrimarySurface->Blt(&rect2, g_pDDSOffscreen, &rect, DDBLT_WAIT, NUI
        if(FAILED(hr))
    {   MessageBox(ghWnd, "Blt failed", "Error", MB_OK);
                    ExitCode();
    }
        }
}

//Renders a file to a multimedia stream
HRESULT RenderFileToMMStream(LPCTSTR szFilename)                //IMultiMediaStrean
{
        HRESULT hr;
        IAMMultiMediaStream *pAMStream=NULL;

//Convert filename to Unicode
        WCHAR wFile[MAX_PATH];
        MultiByteToWideChar(CP_ACP, 0, szFilename, -1, wFile,
                                                            sizeof(wFile)/sizec

        //Create the AMMultiMediaStream object
    hr =CoCreateInstance(CLSID_AMMultiMediaStream, NULL, CLSCTX_INPROC_SERVER,
                                    IID_IAMMultiMediaStream, (void **)&pAMStream)
        if (FAILED(hr))
        {   MessageBox(ghWnd, "Could not create a CLSID_MultiMediaStream object\n"
                "Check you have run regsvr32 amstream.dll\n", "Error", MB_OK);
                  return E_FAIL;
        }

        //Initialize stream
    hr = pAMStream->Initialize(STREAMTYPE_READ, 0, NULL);
        if (FAILED(hr))
        {   MessageBox(ghWnd, "Initialize failed.", "Error", MB_OK);
                  return E_FAIL;
        }
        //Add primary video stream
    hr = pAMStream->AddMediaStream(g_pDD3, &MSPID_PrimaryVideo, 0, NULL);
        if (FAILED(hr))
        {   MessageBox(ghWnd, "AddMediaStream failed.", "Error", MB_OK);
                  return E_FAIL;
        }
        //Add primary audio stream
    hr = pAMStream->AddMediaStream(NULL, &MSPID_PrimaryAudio, AMMSF_ADDDEFAULTRENDE
        if (FAILED(hr))
        {   MessageBox(ghWnd, "AddMediaStream failed.", "Error", MB_OK);
                  return E_FAIL;
        }
        //Opens and automatically creates a filter graph for the specified media f
        hr = pAMStream->OpenFile(wFile, 0);
        if (FAILED(hr))
        {   MessageBox(ghWnd, "File format not supported.", "Error", MB_OK);
                  return E_FAIL;
        }

        //save the local stream to the global variable
        g_pMMStream = pAMStream;
        // Add a reference to the file
        pAMStream->AddRef();

        return NOERROR;
}

HRESULT InitDDrawEx()
{
        HRESULT                 hr=NOERROR;
        DDSURFACEDESC   ddsd, ddsd2;

        CoInitialize(NULL);

        //Create a DirectDrawFactory object
        hr = CoCreateInstance(CLSID_DirectDrawFactory, NULL, CLSCTX_INPROC_SERVER,
                                                    IID_IDirectDrawFactory, (vc
        if (FAILED(hr))
        {   MessageBox(ghWnd, "Couldn't create DirectDrawFactory", "Error", MB_OK)
                  return E_FAIL;
        }
    //Call the IDirectDrawFactory::CreateDirectDraw method to create the
    //DirectDraw object, set the cooperative level, and get the address
    //of an IDirectDraw interface pointer
    hr = (g_pDDF->CreateDirectDraw(NULL, GetDesktopWindow(), DDSCL_NORMAL,
                            NULL, NULL, &g_pDD));

    if (FAILED(hr))
    {   MessageBox(ghWnd, "Couldn't create DirectDraw object", "Error", MB_OK)
            return E_FAIL;
    }

    //Now query for the new IDirectDraw3 interface
    hr =(g_pDD->QueryInterface(IID_IDirectDraw3, (LPVOID*)&g_pDD3));

    if (FAILED(hr))
    {   MessageBox(ghWnd, "Couldn't get IDirectDraw3", "Error", MB_OK);
            return E_FAIL;
    }

//Initialize the DDSURFACEDESC structure for the primary surface
      ZeroMemory(&ddsd, sizeof(ddsd));
ddsd.dwSize = sizeof(ddsd);
      ddsd.dwFlags = DDSD_CAPS;
ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE;
hr = g_pDD3->CreateSurface(&ddsd, &g_pPrimarySurface, NULL);

if(FAILED(hr))
{   MessageBox(ghWnd, "Couldn't create Primary Surface", "Error", MB_OK);
      return E_FAIL;
    }


    // Now, do the same for the offscreen surface.

// The offscreen surface needs to use the same pixel format as the primary.
// Query the primary surface to for its pixel format.
hr = g_pPrimarySurface->GetSurfaceDesc(&ddsd);
if(FAILED(hr))
{   MessageBox(ghWnd, "Couldn't GetSurfaceDesc", "Error", MB_OK);
            return E_FAIL;
}

    // Now, set the info for the offscreen surface, using the primary's pixel
ZeroMemory(&ddsd2, sizeof(ddsd2));
    ddsd2.dwSize = sizeof(ddsd2);
    ddsd2.dwFlags = DDSD_CAPS | DDSD_HEIGHT | DDSD_WIDTH | DDSD_PIXELFORMAT;
ddsd2.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN;
ddsd2.dwHeight = ddsd.dwHeight;     //set the height of the surfaces equal
ddsd2.dwWidth  = ddsd.dwWidth;      //set the width of the surfaces equal
ddsd2.ddpfPixelFormat = ddsd.ddpfPixelFormat; //set the pixel formats equal


// Now, create the offscreen surface and query for the latest interface.
    hr = g_pDD3->CreateSurface(&ddsd2, &g_pDDSOffscreen, NULL);
    if(FAILED(hr))
{   MessageBox(ghWnd, "Couldn't create Offscreen Surface", "Error", MB_OK);
            return E_FAIL;
}


    //Add code for Clipper
    hr = g_pDD3->CreateClipper(0, &g_pDDClipper, NULL);
    if(FAILED(hr))
    {   MessageBox(ghWnd, "Couldn't create Clipper", "Error", MB_OK);
                    return E_FAIL;
    }

        hr = g_pPrimarySurface->SetClipper(g_pDDClipper);
        if(FAILED(hr))
    {   MessageBox(ghWnd, "Call to SetClipper failed", "Error", MB_OK);
                    return E_FAIL;
    }

        hr = g_pDDClipper->SetHWnd(0, ghWnd);
        if(FAILED(hr))
    {   MessageBox(ghWnd, "Call to SetHWnd failed", "Error", MB_OK);
                return E_FAIL;
    }

        return NOERROR;
}

// Display the open dialog box to retrieve the user-selected movie file
BOOL GetOpenMovieFile(LPSTR szName)//LPSTR szName
{
        OPENFILENAME    ofn;

        ofn.lStructSize       = sizeof(OPENFILENAME);
        ofn.hwndOwner         = ghWnd;
        ofn.lpstrFilter       = NULL;
        ofn.lpstrFilter       = "Video (*.avi;*.mpg;*.mpeg)\0*.avi;*.mpg;*.mpeg\0A
        ofn.lpstrCustomFilter = NULL;
        ofn.nFilterIndex      = 1;
        *szName = 0;
        ofn.lpstrFile         = szName;
        ofn.nMaxFile          = MAX_PATH;
        ofn.lpstrInitialDir   = NULL;
        ofn.lpstrTitle        = NULL;
        ofn.lpstrFileTitle    = NULL;
        ofn.lpstrDefExt       = NULL;
        ofn.Flags             = OFN_FILEMUSTEXIST | OFN_READONLY | OFN_PATHMUSTEXI
        return GetOpenFileName((LPOPENFILENAME)&ofn);
}

LRESULT CALLBACK WndMainProc (HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam

   {  // WndMainProc //

        int                                             response;
        HRESULT                                 hr;
        BOOL                                    bOpen;
        static TCHAR                    szFilename[MAX_PATH];
        static STREAM_TIME              StreamTime;             // Stream time of t

        switch(message)
        {
                case WM_COMMAND:
                        {
               switch(wParam)
                          //Program menu option
                          {
                               case IDM_OPEN:
                                       //If a file is already open - call STOP fir
                                       if (g_bAppactive && g_bFileLoaded) {
                                               g_pMMStream->SetState(STREAMSTATE_S
                                       }
                        bOpen = GetOpenMovieFile(szFilename);
                        if (bOpen) {
                                hr = RenderFileToMMStream(szFilenan
                                if (FAILED(hr)) {
                                        ExitCode();
                                        break;
                                        }
                                hr = InitRenderToSurface();
                                if (FAILED(hr)) {
                                        ExitCode();
                                        break;
                                        }
                                g_bAppactive = g_bFileLoaded = TRUI
                                g_bPaused = FALSE;              //T
                                //Now set the multimedia stream to
                                hr = g_pMMStream->SetState(STREAMST
                                if (FAILED(hr))
                                {   ExitCode();
                                }
                        }
                        break;

                case IDM_START:
                        if (g_bAppactive && g_bFileLoaded)
                        {break;                                         //
                        }
                        else {
                                if (g_bPaused) {        // If its i
                                        g_pMMStream->Seek(StreamTim
                                        g_pMMStream->SetState(STREA
                                        g_bAppactive = TRUE;
                                        g_bPaused = FALSE;
                                        }
                                else {

                        if (g_bFileLoaded) {    // If a file is act
                                hr = RenderFileToMMStream(szFilenan
                                if (FAILED(hr)) {
                                        ExitCode();
                                        break;
                                        }
                                hr = InitRenderToSurface();
                                if (FAILED(hr)) {
                                        ExitCode();
                                        break;
                                        }
                                g_bAppactive = g_bFileLoaded = TRUI
                                //Now set the multimedia stream to
                                hr = g_pMMStream->SetState(STREAMST
                                if (FAILED(hr))
                                {   ExitCode();
                                }
                        }
                        else {
                                MessageBox(hWnd, "Please select a n
                        }
                        }
                        }
                        break;

                case IDM_PAUSE:
                        // Pause if not already in a paused state a
                        if (!g_bPaused &&g_bFileLoaded) {
                                hr = g_pMMStream->GetTime(&StreamTi
                                        hr = g_pMMStream->SetState(STREAMST
                                        g_bAppactive = FALSE;
                                        g_bPaused       = TRUE;
                                }
                                break;                          // If its a

                        case IDM_STOP:
                                if (g_bFileLoaded) {
                                        g_pMMStream->SetState(STREAMSTATE_S
                                        StreamTime = 0; // Reset the stream
                                        g_pMMStream->Seek(StreamTime);  //R
                                        g_pMMStream->SetState(STREAMSTATE_R
                                        RenderToSurface();
                                        g_pMMStream->SetState(STREAMSTATE_S
                                        StreamTime = 0;
                                }
                                g_bAppactive = FALSE;
                                break;

                        case IDM_ABOUT:
                                MessageBox(hWnd, "This application uses mul
                                        " render a video file to a DirectDr
                                        "About", MB_OK);
                                break;

                        case IDM_EXIT:
                                response = MessageBox(hWnd, "Quit the Progr
                                if (response==IDYES) SendMessage(ghWnd, WM_
                                break;
        }
                break;
                        }
              break;

    case WM_DESTROY:
                ExitCode();
      break;

              case WM_ACTIVATE:
                      if((BOOL)LOWORD(wParam) == WA_INACTIVE)
                      {
                              //App is not active
                              g_bAppactive = FALSE;
                      }
                      else
                      {
                              //Set app to active if a file is loaded
                              g_bAppactive = (g_bFileLoaded)?TRUE:FALSE;
                      }
                      break;

    default:
      return DefWindowProc(hWnd, message, wParam, lParam);

   }  // Window msgs handling

   return FALSE;

 }  // WndMainProc //

int PASCAL WinMain(HINSTANCE hInstC, HINSTANCE hInstP, LPSTR lpCmdLine, int nCmdSho

 {  // WinMain //
```

```
MSG              msg;
WNDCLASS  wc;
      HRESULT     hr;

ZeroMemory(&wc, sizeof wc);
wc.lpfnWndProc = WndMainProc;
ghInst = wc.hInstance = hInstC;
      wc.hbrBackground = (HBRUSH)GetStockObject(BLACK_BRUSH);
wc.lpszClassName = CLASSNAME;
wc.lpszMenuName = MAKEINTRESOURCE(IDR_MENU);
wc.hCursor = LoadCursor(NULL, IDC_ARROW);
RegisterClass(&wc);

ghWnd = CreateWindowEx(WS_EX_WINDOWEDGE,
          CLASSNAME,
  APPLICATIONNAME,
  WS_VISIBLE |WS_POPUP |WS_OVERLAPPEDWINDOW,
  150,
  150,
  280,
  250,
  0,
  0,
  ghInst,
  0);
    if (ghWnd) {                          // If the call to create window suc
            hr = InitDDrawEx();   // initialize DirectDrawEx
            if (FAILED(hr)) {
                    ExitCode();
            }
    }
    else {
            MessageBox(ghWnd, "Couldn't create window.", "Error", MB_OK);
            return 0;
    }



    ShowWindow(ghWnd, SW_NORMAL);
UpdateWindow(ghWnd);

    while(1){
          //The PeekMessage function checks a thread message queue
          //for a message and places the message (if any) in the specified s
          if(PeekMessage(&msg, NULL, 0,0,PM_NOREMOVE)){

                  // Quit if WM_QUIT found
                  if(!GetMessage(&msg,NULL, 0, 0)) return (msg.wParam);

                  // Otherwise handle the messages
                          TranslateMessage(&msg);         // Allow input
                          DispatchMessage(&msg);          // Send to appropri
          }
          else{
                  // If there are no other windows messages...
                  // Render frame by frame (but only if the App is the active
                  // window and a file is actually loaded)
                  if (g_bFileLoaded && g_bAppactive) {
                          RenderToSurface();
                          }
                  }
          }
    return msg.wParam;
```

```
}  // WinMain //
```

Previous Home Topic Contents Index Next

Control an External Device in DirectShow

This article provides background for developers interested in adding external device control and timecode support to DirectShow applications. It also discusses how timecode is used in the production environment and lists some typical applications that rely on external devices. Finally, it describes how external device control is implemented and provides links to the interfaces available to build VCR control- and timecode-enabled filters in DirectShow.

Contents of this article:

Introduction

You can control an external device in DirectShow by implementing device control filters. These filters control devices or streams of data which are entirely external to the computer and expose interfaces such as IAMExtDevice, IAMExtTransport, IAMTimecodeGenerator, and IAMTimecodeReader. Generally, external device control filters do not need to expose pins. However, an example of a device control filter that does expose pins might be a filter representing a source of data such as a VCR. A pin-to-pin connection representing the data flowing from the VCR to the capture board allows the device control filter and the video capture filter to talk to each other and negotiate data types, athough they do not use the standard transport and no data would flow between the filters themselves other than control information. Applications can instantiate and directly control an external device filter, but it is strongly recommended that they are always instantiated within the context of a filtergraph, even if they are the only filter in the graph.

External devices can include VCRs, video editing stations, audio tape recorders (ATRs), mixers, or any other device used in the video capture and editing process. Capture and editing requires DirectShow external device control filters to provide audio and video synchronization and precise control. You can accomplish synchronization of audio and video during playback, edit, and capture with external clocks or Society of Motion Picture and Television Engineers (SMPTE) timecode. Understanding timecode is the key to understanding external device control.

Understanding SMPTE Timecode

SMPTE timecode is the glue that holds the post-production process together. It identifies video and audio sources, makes automatic track synchronization possible, and provides a container for ancillary data related to the production. You will need to understand this data stream and its application to media production, tool development, or system design.

SMPTE timecode, more properly known as SMPTE time and control code, is a series of digital frame address values, flags and additional data applied to a video or audio stream, and is defined in ANSI/SMPTE 12-1986. Its purpose is to provide a machine readable address for video and audio.

The most common form of a SMPTE timecode data structure an 80-bit frame that contains the following information:

1. A time stamp in hh:mm:ss:ff (hours:minutes:seconds:frames) format.
2. Eight 4-bit binary groups commonly known as userbits.
3. Various flag bits.
4. Synchronization sequence.
5. Checksum.

The DirectShow TIMECODE_SAMPLE structure is an example of a timecode data structure that contains timecode information for video or audio data.

SMPTE timecode comes in one of two types. Timecode recorded on an analog audio track as a bi-phase mark encoded signal is known as LTC, or Linear TimeCode (formerly known as Longitudinal TimeCode). Each timecode frame is one video frame time in duration. The other common type of timecode is known as VITC, or Vertical TimeCode. VITC is usually stored on two lines of a video signal's vertical blanking interval, somewhere between lines 10 and 20.

LTC timecode is easy to add to a pre-recorded tape, since it is encoded in a separate audio signal. However, it cannot be read when the tape is paused, moving very slowly, or very quickly. In addition it consumes one audio channel on non-professional VCRs.

VITC timecode, on the other hand, can be read from speeds of zero to 15 times normal speed. It can contain field-dependent data and can be read from video capture cards. However, it is not easily added to a prerecorded tape and often requires expensive hardware.

SMPTE timecode also comes in one of two modes, non-drop frame and drop frame. Non-drop frame is timecode that is consistently increasing and sequential. It can act as a real-time clock and works fine for monochrome video that runs at a frame rate of exactly 30 frames per second.

However, NTSC color video actually runs at a frame rate of 29.97 Hz (frames per second) because of some compatibility issues with monochrome television. This causes a problem with non-drop frame timecode because it gets out of step with real-time at the rate of 108 frames (or 3.6 seconds) per hour. This means that after 1 hour of playback, the timecode would read 00:59:56:12, assuming a start point of 00:00:00:00. This causes problems when trying to calculate show duration or using "time-of-day" referencing.

A solution to this problem is to skip some frames in the count every so often so the error is reduced to something tolerable. This compensation method is called "drop frame" and is implemented by skipping the first two frames from the count at the start of each minute except minutes 00, 10, 20, 30, 40 and 50. The net result is an error of less than 1 frame per hour, or about 3 frames per 24 hour period.

Drop frame is used more commonly in today's productions, although any implementation should support mixing both modes.

Typical Uses of Timecode

Applications which provide video capture and editing functionality will typically require control of external devices. These applications need to identify and index video and audio frames through references to SMPTE timecode. Linear editing system computers generally control three or more tape machines, as well as a video switcher and possibly a digital disk recorder. The controlling computer must execute commands at precise times and therefore must get videotapes cued to specific places at specific points in time.

Applications typically use timecode in a number of different ways including, but not limited to the following:

- Tracking of video and audio sources throughout the editorial process so an edit decision list, or EDL, may be generated for archival or export to another system. To create an EDL:
  1. Shoot the video.
  2. Capture into a nonlinear offline system that uses some form of intraframe-only compression (MJPEG, DV, etc.).
  3. Edit the material and generate an edit decision list (EDL) and offline edited master.
  4. Import the EDL to an online system and do an "auto-assembly" using the original source material to generate the final master, adding titles and effects where required.
- Synchronizing audio to video. In feature film production, audio is usually recorded on a separate tape recorder along with timecode. Specially equipped film cameras can also record timecode on the film in between the sprocket holes. After the filmed image is electronically transferred to videotape, the timecode is used to align the audio with the picture in a process known as "synching the dailies". If the audio and video timecodes are different, VITC and LTC may sometimes be used together, one for video timecode and the other for audio timecode.
- Synchronization and triggering of multiple devices such as ATRs, digital disk recorder or players, VCRs, or other similar devices. This is a much broader class of synchronization than described above, and is most commonly seen in linear editing and nonlinear editing systems, closed captioning systems, and subtitling systems.
- Making use of the undefined bits in the timecode, called userbits. Often information such as dates, ASCII codes, or film industry information is contained in the userbits, however uses of userbits is limited only to the ingenuity of the user.

It quickly becomes obvious that timecode makes many things possible when properly handled. Unfortunately, there is also a lot that can go wrong, either because of poor technique or hardware malfunctions. Some things to look out for on timecoded tapes are:

1. Unstable or drifting timecode relative to video or audio.
2. Poor timecode field integrity. This means an LTC word begins in the middle of a frame rather than at the beginning, or VITC is not updated on a true frame boundary. The net result is an ambiguous reference.
3. Unintentional VITC/LTC mismatch.
4. Intermittent dropouts.
5. Missing timecode.

6. Poor timecode signal quality.
7. Incremental frame offset from incorrectly made copies.

Capturing Timecode

Timecode can be generated either by an external timecode generator, by a capture card capable of generating timecode, by the device control filter itself, or by an external device such as a VCR that has a built-in timecode reader. An RS-422 connection is generally necessary if the timecode is sent to the host from an external device.

Once timecode is generated, it needs to be captured either in tabular or stream format concurrently with the video or audio so that it can later be accessed during editing. This is handled in one of two ways:

1 Build a table that lists the timecode discontinuities indexed to frame position within the stream, and write the table to the end of the file after capture is complete. The list might be an array of structures that look like this (NOTE, the following structure is a simplification of the DirectShow TIMECODE_SAMPLE structure and is intended as an example only):

```
struct  {
                DWORD dwOffset; // offset into stream in frames
                char[11] szTC;          // timecode value at offset in hh:mm:ss:ff
                                                //  for non drop, hh:mm:ss;ff for dr
                } TIMECODE;
```

For example, given a captured video stream with one timecode break in it, the list might look like this:

```
{0, 02:00:00:02},
{16305, 15:21:13:29}            // timecode jumps at frame 16305
```

Using this table, any frame's timecode can be easily calculated.

2 Treat the data as a stream and write it to the file just as video and audio are written. This is useful for rapidly changing data or even non-timecode data in the vertical blanking interval (VBI) such as closed captioning data.

Once the timecode data is properly stored with its associated frame data, applications that edit, composite, synchronize or trigger can access and use a familiar and standard indexing system.

External Device Control

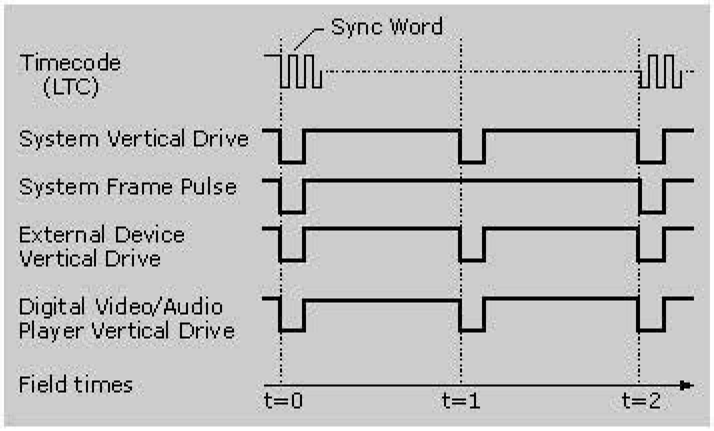
To understand external device control, it is necessary to understand timecode. The key things to remember about timecode are:

- SMPTE timecode is a frame addressing system that identifies video and audio frames. It comes in many types and modes: LTC, VITC, Drop Frame, Non-Drop Frame, and operates at various frame rates: 24, 25, 29.97, and 30 frames per second.
- SMPTE timecode is used in edit decision lists (EDLs) which are generated for offline editing and online editing, as a timing reference for synchronizing hardware devices, and as a vehicle for additional data such as production source or film reference information.
- SMPTE timecode can be stored as a stream or table of discontinuities.
- Timelines are necessary for synchronization, and can be local to the controlling

computer, external synchronizer, or the controlled device itself.

Given this background, two fundamental problems exist with device control. First, hundreds of different communication protocols exist for all the various devices from all the various manufacturers. Although some devices are more widely used than others, such as VCRs and Laserdiscs, almost all have a different remote control interface. As more sophisticated professional video and audio applications continue to move to the desktop, this problem gets worse. Due to this myriad of protocols, separate DirectShow filters must be implemented for each and every external device you want to control.

Second, the fundamental problem in the design of professional video and audio systems is that events must occur at precise points in time. Taking a systems view of this issue, consider the following timing diagram:

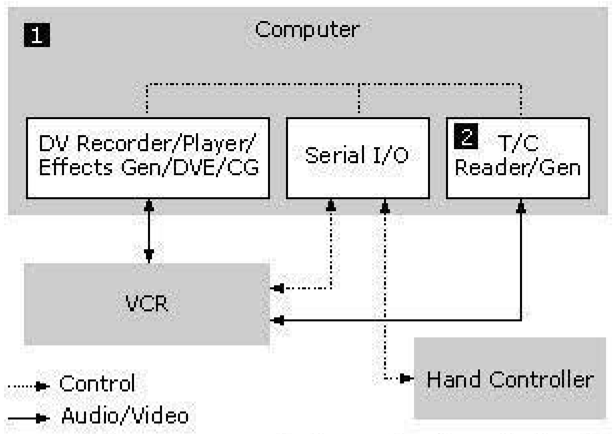


The horizontal axis denotes time in video fields, or roughly 1/60 of a second for NTSC video. The key point here is that all signals line up in time, that is, timecode starts at the beginning of a frame (System Frame Pulse). External devices such as tape machines are aligned with the system reference, as well as digital video playback such as an AVI file run from an AVI-enabled application.

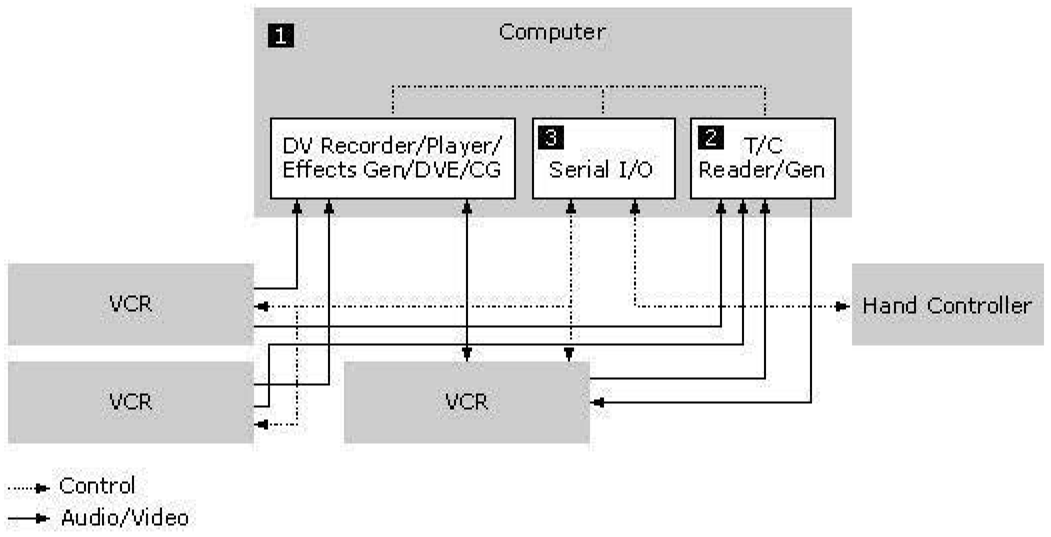
Conformance to this timing requirement is achieved by various means, the most common of which is a master reference signal distributed to all components in the system. This reference is known as "blackburst" in the video world, so named because it is a composite video signal containing no active video above black level. The "burst" portion of the name refers to the color burst portion of the video signal. Each device connected to the reference is responsible for maintaining its own synchronization. This means for example, that a digital video player must switch frames during the vertical blanking interval, a tape machine must switch into record mode during the vertical blanking interval, commands sent to external devices via a serial port must be timed to the frame pulse, and all of these and other synchronized events must occur when the SMPTE timecode hits a predetermined value. Failure to conform to these rules results in tearing of a video image or edits occurring at the wrong point in time.

Accomplishing all this in the professional video world is relatively straightforward, but in the hybrid world of desktop video, it is very difficult.

Building on the concepts presented so far, the two design examples in the following diagrams illustrate a potential configuration of external devices.



Notes: 1 Computer Monitor/Mouse/Keyboard omitted for clarity.
2 Timecode reader required for non-timecode-enabled VCRs.



Notes: 1 Computer/Monitor/Mouse/Keyboard omitted for clarity.
2 Timecode reader required for non-timecode-enabled VCR.
3 Serial I/O can be replaced by a more sophisticated control subsystem (plug-in or external).

The block diagrams show that it is relatively simple to distribute the reference signal to all of the boxes. To deal with synchronization that takes place within the computer, for example, between the timecode reader and digital video player, it is recommended that either a "vertical drive" hardware interrupt, specialized operating system services, or some other custom solution be used.

Finally, if you intend to write an external device filter, you should implement the IAMExtDevice, IAMExtTransport, IAMTimecodeReader, IAMTimecodeGenerator, and IAMTimecodeDisplay interfaces provided by DirectShow. Additionally if you need to move binary messages to and from an external device, for example, to download executable code for the external device's microprocessor to execute, this should be accomplished by implementing the COM IDataObject interface, which has a complete set of methods for handling binary data transfers. Use this interface for whatever custom data transfer purposes your filter needs.

For sample code that demonstrates how to implement an external device control filter see the Samples\DS\Vcrctrl folder in the DirectX Media SDK.

References and Suggested Reading

For additional information on SMPTE timecode and external device control, refer to the following documentation.

1. Proposed revision to ANSI/SMPTE 12M-1986, SMPTE Standard for Television, Audio and Film Time and Control Code, SMPTE Journal, February 1995.
2. SMPTE RP 135-1990 "Use of Binary User Groups in Motion-Picture Time and Control Codes"
3. SMPTE RP 169 "Television, Audio and Film Time and Control Code - Auxiliary Time Address Data in Binary Groups - Dialect Specification of Directory Index Locations"
4. SMPTE RP 179-1994 "Dialect Specification of Page-Line Directory Index for Television, Audio and Film Time and Control Code for Video-Assisted Film Editing"
5. "Touring the Vertical Interval", Warner Johnston, TV Technology, August 1991
6. "Closed Captioning in Real Time", Marc Oakrand, SMPTE Journal, June 1991
7. Timecode Handbook, 3rd. Edition, Cipher Digital, Inc. (available from Mix Bookshelf)
8. Timecode: A Users Guide, John Ratcliffe (available from Mix Bookshelf)
9. SMPTE RP 138 - Control Message Architecture
10. SMPTE RP 139 - Tributary Interconnection
11. SMPTE RP 163 - System Service Messages
12. SMPTE RP 170 - Video Tape Recorder Type Specific Messages for Digital Control Interface
13. SMPTE RP 171 - Type-Specific Messages for Digital Control Interface of Analog Audio Tape Recorders
14. SMPTE RP 172 - Common Messages for Digital Control Interface
15. SMPTE 275M - ESlan-1 Remote Control System

Note SMPTE standards and reprints available from SMPTE at (914)761-1100

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

Build a Filter or Application with Visual C++ 5.x

This article describes step-by-step procedures for building your Microsoft® DirectShow™ projects. You can either build your filter or application from the command line, or build it from within the Visual Studio environment that comes with Microsoft Visual C++®. If you choose to build a filter from within Microsoft Visual Studio™, you can use the VC5Kit set of files shipped with the DirectShow SDK, by default, in the dxmedia\tools\VC5Kit directory. This provides an easy way to configure Visual Studio project settings. You can also set the project settings within Visual Studio yourself.

The main difference between building a filter and building an application from within Visual Studio is that for a filter you select **Win32 Dynamic-Link Library** as the project type, while for an application you select **Win32 Application** or **Win32 Console Application** as the project type. (You choose the project type in the New Projects dialog box opened by choosing the **New** command from the **File** menu.)

This article includes the following topics.

- Building a Filter or Application from the Command Line
- Using VC5Kit to Build a Filter in Visual Studio
- Setting DirectShow Project Settings in Visual Studio

Building a Filter or Application from the Command Line

Perform the following steps to build a DirectShow project from the command line.

1. Go to the Visual C++ Bin directory.
2. Type `VCVARS32`.
3. Go to the directory containing the filter you want to build.
4. If you are building a sample filter and the sample filter isn't in the default directory (the default is \dxmedia\Samples\ds\Samplename; for example, \dxmedia\Samples\ds\Gargle), set the AXSDK MAKEFILE variable to the top-level DirectShow directory. For example, at the command prompt, type:

   ```
   set axsdk=c:\dxmedia
   ```

5. At the command prompt, type:

   ```
   NMAKE
   ```

   For nondebug versions, type:

   ```
   NMAKE NODEBUG=1
   ```

Using VC5Kit to Build a Filter in Visual Studio

This section discusses building filters with Microsoft® Visual C++® version 5.*x*. The 5.*x* VC5Kit is installed by default in the dxmedia\tools\VC5Kit directory.

- Setting the Visual Studio Include and Lib Directories
- Creating a Project Directory and Adding Source Files
- Building the Project in Visual Studio
- Creating a GUID
- Creating a Definition File

Setting the Visual Studio Include and Lib Directories

This topic describes how to set your Visual Studio Include and Lib directories. If you are building a DirectShow sample application, such as PlayFile, that comes with a makefile or .mak file, the steps in this topic are the only ones you must perform to build the sample. Once you set your directories, you can choose **Open Workspace** from the **File** menu and select the existing makefile or .mak file. Visual Studio will wrap the makefile. If you are building a filter, other steps will probably be necessary.

1. Open Visual Studio.

2. From the **Tools** menu, choose **Options**.
3. Choose the **Directories** tab.
4. In the **Show directories for** drop-down list, select **Include files**.
5. Add the DirectShow include directories (by default: dxmedia\Include and dxmedia\Classes\Base) to the list.
6. In the **Show directories for** drop-down list, select **Library files**.
7. Add the DirectShow library directory (by default C:\dxmedia\Lib).

Creating a Project Directory and Adding Source Files

Follow these steps to create your project directory and add your source files to it:

1. Create an empty directory for your project; for example, C:\Filter.
2. Copy the VC5Kit Filter.def, Filter.dsp, Filter.dsw, and Filter.mak files into the directory.
3. Copy your source files into the directory. This includes .h, .cpp and any other miscellaneous source files your project requires. If your project has its own .def and .rc files, you can use these rather than the .def and .rc files provided with VC5Kit.

Building the Project in Visual Studio

Follow these steps to build your project in Visual Studio:

1. From the **File** menu in Visual Studio, choose **Open Workspace**.
2. In the **Open Workspace** dialog box that appears, browse to the directory you created and select Filter.dsw.

   The project opens.

3. To add your source files to the project, choose **Add to Project** from the **Project** menu, and then choose **Files** from the submenu that appears. Browse to the directory containing your source files, select the ones you want to add, and click the **OK** button.
4. From the **Project** menu, choose **Settings**.
5. Choose the **Link** tab and select **General** from the **Category** drop-down list.
6. In the **Output** file name box, type the name of your filter; for example, Filter.ax.
7. Choose **OK** to confirm the project settings you've selected.
8. Choose **Build** *Filter.ax* from the **Build** menu. The name *Filter.ax* is the name you gave the output file in the **Link** tab.

Creating a GUID

If you are building your filter using source files from an existing filter, including any samples filter that ships with DirectShow, you must create a GUID for the new filter.

To create a GUID in Visual Studio:

1. From the **Tools** menu, choose **Create GUID**. By default, the GUID is in DEFINE_GUID format, which is the format you want.
2. Click the **Copy** button.
3. Delete the old GUID from your source file.
4. Put the cursor in your source file where the old GUID used to be, and choose **Paste** from the **Edit** menu.

Creating a Definition File

If your filter implements the DllGetClassObject, DllCanUnloadNow, DllRegisterServer, or DllUnregisterServer functions, you must include a definition file (.def file) that exports them. For example (where *Filter.ax* is the name you gave the output file):

```
LIBRARY FILTER.AX
DESCRIPTION 'Description of my filter'
PROTMODE
EXPORTS
    DllGetClassObject
    DllCanUnloadNow
    DllRegisterServer
    DllUnregisterServer
```

Setting DirectShow Project Settings in Visual Studio

This section describes how to set project settings in Visual Studio to build your own DirectShow applications and filters. If you are building samples supplied by DirectShow, you need only set your Include and Lib directories.

In some cases, you might need to add to these project settings for your particular application. For example, if you use DirectDraw functions, you must add Ddraw.lib to the list of Link libraries.

In all cases, you must set the Visual Studio Include and Lib directories as described in Setting the Visual Studio Include and Lib Directories.

This section contains the following topics.

- Creating a Project
- Adding Files to the Project
- Setting Project Settings For Both Release and Debug Builds
- Setting Project Settings for Debug Builds
- Building a Release or Debug Version of Your Project

Creating a Project

To create a project, perform the following steps.

1. From the **File** menu, choose **New**.
2. Choose the **Projects** tab.
3. If you are building an application, select **Win32 Application** as the type of project. If you are building a filter, select **Win32 Dynamic-Link Library**.
4. Type a name for the project and a location for the project files.

Adding Files to the Project

To add files to the project, perform the following steps.

1. From the **Project** menu, choose **Add to Project**. From the submenu that appears, choose **Files**.
2. In the **Insert Files into Project** dialog box that appears, browse for the filter files you want to add to the project, such as the .cpp, .h, .rc, and .def files.

3. Select the file or files you want to add and choose **OK**.

Setting Project Settings For Both Release and Debug Builds

To set project settings for both release and debug builds, perform the following steps.

1. From the **Project** menu, choose **Settings**.
2. In the dialog box that appears, select **All Configurations** in the **Settings For** drop-down list.

Follow these steps to set your project general settings:

1. From the **Project** menu, choose **Settings**.
2. Choose the **General** tab.
3. From the **Microsoft Foundation Classes** drop-down list, select **Not Using MFC**.

Follow these steps to set your project compiler settings:

1. From the **Project** menu, choose **Settings**.
2. Choose the **C/C++** tab.
3. In the **Category** drop-down list, select **General**.
4. In the **Preprocessor definitions** text box, insert the following:

   ```
   INC_OLE2,STRICT,WIN32,_MT,_DLL,_X86_=1,WINVER=0x0400
   ```

5. Select the **C++ Language** category.
6. Select the **Enable exception handling** check box.
7. Choose the **Code Generation** category.
8. In the **Processor** drop-down list, select **Blend***.
9. In the **Calling convention** drop-down list, select **_stdcall**.
10. In the **Use Run-time library** drop-down list, select **Multithreaded DLL**.
11. Select the **Precompiled Headers** category.
12. Select the **Not using precompiled headers** option button.

Follow these steps to set your project link settings:

1. From the **Project** menu, choose **Settings**.
2. Choose the **Link** tab.
3. In the **Category** drop-down list, select **General**.
4. In the **Output file name** text box, type the filter's output file name; for example, Debug/Filter.ax.
5. Add the following libraries to the beginning of the **Object/Library modules** text box.

   ```
   quartz.lib strmbase.lib msvcrt.lib
   ```

   These libraries must be the first libraries in the link list. Depending on the functions your application accesses, you might need to add other libraries to this list.

6. Select the **Ignore all default libraries** check box.
7. Select the **Customize** category.
8. Clear the **Use program database** check box.
9. Select the **Output** category.
10. In the **Base address** text box, type:

```
0x1c400000
```

11. In the Entry-point symbol text box, type:

```
DllEntryPoint@12
```

Setting Project Settings for Debug Builds

To set project settings for debug builds, perform the following steps.

1. From the **Project** menu, choose **Settings**.
2. Select **Win32 Debug** in the **Settings For** drop-down list in the **Project Settings** dialog box that appears.
3. Choose the **C/C++** tab.
4. In the **Category** drop-down list, select **General**.
5. Select the **Generate browse info** check box.
6. In the **Debug info** drop-down list, select **C7 Compatible**.
7. Select the **Code Generation** category.
8. In the **Use Run-time library** drop-down list, select **Debug Multithreaded DLL**.
9. Choose the **Link** tab.
10. Select the **Debug** category.
11. Select the **Debug info** check box.

Building a Release or Debug Version of Your Project

To build a release or debug version of your project:

1. Choose **Set Active Configuration** from the **Build** menu, and select **Win32 Release** or **Win32 Debug** from the list that appears.
2. Choose **Build** *Filter.ax* from the Build menu. The name *Filter.ax* is the name you gave the output file in the **Link** tab.

Previous Home Topic Contents Index Next

Recompress an AVI File

The following sample code shows how to recompress the file c:\Foo.avi to c:\Bar.avi, where the output file will use Cinepak compression and will include CD-quality audio. Recompression is useful to change the format of a file from one compression scheme to another. The exact benefits of recompression depend on the different compressors used to compress the source and output files and often include producing a smaller output file. In this example the source file, Foo.avi, might be in a format such as uncompressed RGB with 22 kilohertz (kHz) sound.

The AMCap Sample (DirectShow Capture Application) sample demonstrates a capture application and uses many of the same concepts as the following code.

Note This sample code fragment introduces concepts only and is not designed to compile. See the AMCap Sample (DirectShow Capture Application) sample for actual code. The code fragment does not perform error checking for the sake of brevity.

```
    // Create a graph builder object.
    hr = CoCreateInstance((REFCLSID)CLSID_CaptureGraphBuilder,
                 NULL, CLSCTX_INPROC, (REFIID)IID_ICaptureGraphBuilder,
                 (void **)&pBuild);

    // Create a filter graph, and tell the builder what it is.
    hr = CoCreateInstance((REFCLSID)CLSID_FilterGraph,
                 NULL, CLSCTX_INPROC, (REFIID)IID_IGraphBuilder,
                 (void **)&pFg);
    hr = pBuild->SetFiltergraph(pFg);

    // Obtain a source for c:\foo.avi.
    hr = CoCreateInstance((REFCLSID)CLSID_AsyncReader,
                 NULL, CLSCTX_INPROC, (REFIID)IID_IBaseFilter,
                 (void **)&pSrc);
    hr = pSrc->QueryInterface(IID_IFileSourceFilter, (void **)&pI);
    hr = pI->Load(L"c:\\foo.avi", NULL);
    pI->Release();
    hr = pFg->AddFilter(pSrc, NULL);

    // Create a rendering section to create the c:\bar.avi output file.
    hr = pBuild->SetOutputFileName(&MEDIASUBTYPE_Avi, L"c:\\bar.avi", &pRender,
                                   NULL);

// [...Enumerate the audio compressors with the category CLSID_AudioCompressorCateg
// and pick one. See the Amcap.cpp file in the capture sample directory for an exam
// how to enumerate a category...]

    // Render the recompressed audio stream
    hr = pBuild->RenderStream(NULL, pSrc, pCAud, pRender);

// [...Enumerate a video compressor using the CLSID_VideoCompressorCategory enum,
// as seen in Amcap.cpp...]

    pCVid = [...IBaseFilter pointer of the chosen Cinepak compressor...]
    hr = pFg->AddFilter(pCVid, NULL);

    // Tell it to compress at 100k/second data rate.
    // Use the current format to set the data rate, but change the data
    // rate item in the media type.
    hr = pBuild->FindInterface(NULL, pCVid, IID_IAMStreamConfig,
                               (void **)&pVSC);
    hr = pVSC->GetFormat(&cmt);
    ((VIDEOINFOHEADER)(cmt.Format()))->dwBitRate = 100000;
    hr = pVSC->SetFormat(&cmt);
    pVSC->Release();

    // Request key frames every 4 frames.
    hr = pBuild->FindInterface(NULL, pCVid, IID_IAMVideoCompression,
                               (void **)&pVC);
    hr = pVC->put_KeyFrameRate(4);
    pVC->Release();

    // Render the recompressed video stream.
    hr = pBuild->RenderStream(NULL, pSrc, pCVid, pRender);
    // All done with these objects now.
    pSrc->Release();
    pRender->Release();
    pCAud->Release();
    pCVid->Release();

    // Run the graph.
    hr = FG->QueryInterface(IID_IMediaControl, &MC);
    MC->Run();

    // Wait for EC_COMPLETE, and it's all done!
    // [...wait for EC_COMPLETE event... see AMCap sample...]

 pGraphBuilder->FindInterface(pRender, IID_IMediaSeeking, pMS);
// [...While waiting for complete, use IMediaSeeking methods to get
// the percentage complete... IMediaSeeking->GetCurrentPosition divided by
// GetDuration * 100 will tell you the percent complete at any time...]

    pFg->Release();
    pBuild->Release();
```

Register DirectShow Objects

This section describes the steps you must take to make your Microsoft® DirectShow™ objects self-registering. It describes the relationships between the registry entry points called by COM, the globally-defined CFactoryTemplate array elements, and the AMOVIESETUP_MEDIATYPE, AMOVIESETUP_PIN, and AMOVIESETUP_FILTER structures.

To enable objects in a dynamic-link library (DLL) to register themselves, two COM-defined entry points must be provided in the DLL and exported:

- DllRegisterServer
- DllUnregisterServer

With these entry points in your DLL, you can use the Regsvr32.exe tool to register and unregister your DLL or setup tools, or applications can register the filter programmatically.

Implementing Self-Registration

To implement a self-registering filter, carry out the following steps.

1. Add DllRegisterServer and DllUnregisterServer to the export list in your filter's DEF file.
2. Provide implementations for these functions, which call the DirectShow AMovieDllRegisterServer2 function with parameters of TRUE and FALSE, respectively. For example:

```
STDAPI DllRegisterServer()
{
  return AMovieDllRegisterServer2(TRUE);
}

HRESULT DllUnregisterServer()
{
  return AMovieDllRegisterServer2(FALSE);
}
```

You can add code to these functions to set up custom registry entries.

3. Define the setup data structures for each filter based on the AMOVIESETUP_MEDIATYPE, AMOVIESETUP_PIN, and AMOVIESETUP_FILTER structures.

   For example, here are the structures for the Ball.ax sample filter:

```
 // Setup data

const AMOVIESETUP_MEDIATYPE sudOpPinTypes =
{ &MEDIATYPE_Video
, &MEDIASUBTYPE_NULL };

const AMOVIESETUP_PIN sudOpPin =
{ L"Output"
, FALSE
, TRUE
, FALSE
, FALSE
, &CLSID_NULL
, NULL
, 1
, &sudOpPinTypes };

const AMOVIESETUP_FILTER sudBallax =
{ &CLSID_BouncingBall
, L"Bouncing Ball"
, MERIT_UNLIKELY
, 1
, &sudOpPin };
```

4. In the CFactoryTemplate g_Templates array that instantiates your class, ensure that the first parameter has the name of the filter, (for example, "Bouncing Ball") and that the last parameter has the address of the AMOVIESETUP_FILTER structure you defined.

```
CFactoryTemplate gTemplates[]={
 L"Bouncing Ball",                // Name of the filter
 &CLSID_BouncingBall,             // CLSID of the filter
 CBouncingBall::CreateInstance,   // Static function to be called by class f
 NULL,                            //
 &sudBallax}                      // Address of the AMOVIESETUP_FILTER struc
};
```

5. Tag the DLL file as self-registering by adding the string "OLESelfRegistering" to its resource (defining AMOVIE_SELF_REGISTER in your resource file does this automatically if you are using Activex.rcv and Activex.ver). This string enables applications to

determine whether the object is self-registering without loading the DLL.

Enumerate and Access Hardware Devices in DirectShow Applications

This article explains and demonstrates how to initialize and access system hardware devices by using interfaces and classes provided by Microsoft® DirectShow. Developers need this functionality to support many types of hardware in their applications. Typically, DirectShow applications use the following types of hardware.

- Audio and video capture cards
- Audio or video playback cards
- Audio or video compressors or decompressors (such as an MPEG decoder)

Because developers support these devices in a similar manner (and for the sake of brevity), they will be referred to as AV devices for the remainder of this article; they will be distinguished only if a topic applies to a specific type of device.

Three interfaces apply to hardware device support: ICreateDevEnum (documented in the DirectShow SDK), and **IPropertyBag** and **IPersistPropertyBag** (both Microsoft Win32® interfaces). These interfaces handle hardware device enumeration and the loading and storage of AV device properties.

Contents of this article:

Application developers who want to control hardware devices should be familiar with the COM-based concepts of monikers, enumerators, and the initialization and creation of DirectShow objects.

How to Enumerate Hardware Devices

Microsoft provides audio and video capture and playback functionality through interfaces, classes, and samples included in the DirectShow SDK. Because the File Source filters and the filter graph manager handle the internal work of passing information from component to component, adding capture capabilities to an application requires a relatively small amount of additional code. The required additional code enumerates the system's hardware devices and compiles a list of the devices that can perform a specific task (a list of all video capture cards, for example). You can use the same enumeration process for any hardware device, past or present; DirectShow automatically instantiates filters for both Win32 and Video for Windows devices.

To work with AV devices, you must first detect what devices exist on the current system. The ICreateDevEnum interface, which creates enumerators for any specified type of object, provides the functionality you need to detect and set up the hardware. Accessing a specific device is a three-step process, detailed by the following instructions and code fragments.

1. Create a system hardware device enumerator.

   First, set aside a pointer for the enumerator, and then create it by using the CoCreateInstance function; CLSID_SystemDeviceEnum is the type of object you want to create (a system hardware device enumerator, in this case) and IID_ICreateDevEnum is its interface GUID.

   ```
   ICreateDevEnum  *pCreateDevEnum ;
   CoCreateInstance(CLSID_SystemDeviceEnum, NULL, CLSCTX_INPROC_SERVER,
                    IID_ICreateDevEnum, (void**)&pCreateDevEnum) ;
   ```

2. Create an enumerator for a specific type of hardware device (such as a video capture card).

   Declare an IEnumMoniker interface pointer and pass it to the ICreateDevEnum::CreateClassEnumerator method, called on the system device enumerator. You can then use the **IEnumMoniker** interface pointer to access the newly created enumerator.

   ```
   IEnumMoniker *pEnumMon ;
   pCreateDevEnum->CreateClassEnumerator(
                              [specify device GUID here]
                              &pEnumMon, 0);
   ```

3. Enumerate the list itself until you locate the desired device.

   If the previous call to CreateClassEnumerator succeeded, you can call the IEnumMoniker::Next method to step through the list of devices. To retrieve the device itself, call the IMoniker::BindToObject method on an enumerated device. BindToObject creates the filter associated with the selected device and loads the filter's properties (CLSID, FriendlyName, and DevicePath) from the registry. Don't be confused by the (1 == cFetched) portion of the **if** condition; the Next method will set it to the number of returned objects (1, if successful) before testing the statement's validity.

   ```
   ULONG cFetched = 0;
   IMoniker *pMon ;

   if (S_OK == (pEnumMon->Next(1, &pMon, &cFetched))  &&  (1 == cFetched))
   {
       pMon->BindToObject(0, 0, IID_IBaseFilter, (void **)&[desired interface
   ```

   Now that you have the IMoniker pointer, you can add the device's filter to the filter graph. Once you've added the filter, you don't need the **IMoniker** pointer, device enumerator, or system device enumerator.

```
                pGraph->AddFilter([desired interface here], L"[filter name here]"
                pMon->Release() ;  // Release moniker
            }
            pEnumMon->Release() ;   // Release the class enumerator
        }
        pCreateDevEnum->Release();
```

Device Enumeration in the AMCap Sample

The DirectShow SDK includes an audio and video capture sample application called AMCap, as well as the sample's source code. Internally, AMCap uses the ICreateDevEnum interface to construct a list of a system's capture devices. In the application itself, you can access the list of devices from the Devices menu.

The code that builds AMCap's enumerated list of devices is its InitCapFilters function. This function demonstrates a typical way to enumerate filters, for both former and current hardware devices. For the sake of brevity, the following code walk-through contains no error-checking code; for the complete version, see the Amcap.cpp file in the \Samples\DS\Capture directory of the SDK. The AMCap sample uses a global variable, gcap, which is a structure from the Amcap.cpp file that stores a variety of information used by the filter graph. While you generally want to avoid using global variables, this structure does show the amount of information that the filter graph manager handles.

```
struct _capstuff {
    char szCaptureFile[_MAX_PATH];
    WORD wCapFileSize;
    ICaptureGraphBuilder *pBuilder;
    IVideoWindow *pVW;
    IMediaEventEx *pME;
    IAMDroppedFrames *pDF;
    IAMVideoCompression *pVC;
    IAMVfwCaptureDialogs *pDlg;
    IAMAudioStreamConfig *pASC;
    IAMVideoStreamConfig *pVSC;
    IBaseFilter *pRender;
    IBaseFilter *pVCap, *pACap;
    IGraphBuilder *pFg;
    IFileSinkFilter *pSink;
    BOOL fCaptureGraphBuilt;
    BOOL fPreviewGraphBuilt;
    BOOL fCaptureGraphRunning;
    BOOL fPreviewGraphRunning;
    BOOL fCapAudio;
    int  iVideoDevice;
    int  iAudioDevice;
    double FrameRate;
    BOOL fWantPreview;
    long lCapStartTime;
    long lCapStopTime;
} gcap;
```

InitCapFilters starts by defining some basic return and error-checking variables. AMCap uses the uIndex value to loop through the system's hardware devices later.

```
BOOL InitCapFilters()
{
    HRESULT hr;
    BOOL f;
    UINT uIndex = 0;
```

The MakeBuilder function call creates a filter graph builder. You can find the MakeBuilder function in Amcap.cpp.

```
    f = MakeBuilder();
```

The next section handles the video capture device enumeration; this code is very similar to the code description from the How to Enumerate Hardware Devices section. It first declares an ICreateDevEnum pointer, then uses CoCreateInstance to create an enumerator for system hardware devices.

```
    ICreateDevEnum *pCreateDevEnum;
    hr = CoCreateInstance(CLSID_SystemDeviceEnum, NULL, CLSCTX_INPROC_SERVER,
                          IID_ICreateDevEnum, (void**)&pCreateDevEnum);
```

After it has a device enumerator, AMCap creates an enumerator specifically for video capture devices by passing the CLSID_VideoInputDeviceCategory class identifier to ICreateDevEnum::CreateClassEnumerator. It can now use the IEnumMoniker pointer to access the enumerated list of capture devices.

```
    IEnumMoniker *pEm;
    hr = pCreateDevEnum->CreateClassEnumerator(CLSID_VideoInputDeviceCategory, &pEm
    pCreateDevEnum->Release();                  // We don't need the device enumera
    pEm->Reset();                                           // Go to the start
```

Now AMCap needs the actual device; it calls IEnumMoniker::Next to move through the device list, and then points pM to each device by calling IMoniker::BindToObject, which also loads the device's properties (CLSID, FriendlyName, and DevicePath) from the registry. If you do not want to automatically create the filter associated with the device, use IMoniker::BindToStorage instead of BindToObject.

```
    ULONG cFetched;
    IMoniker *pM;                                     // This will access the act
    gcap.pVCap = NULL;
    while(hr = pEm->Next(1, &pM, &cFetched), hr==S_OK)
    {
if ((int)uIndex == gcap.iVideoDevice) { // This is the one we want.  Instantiate it
        hr = pM->BindToObject(0, 0, IID_IBaseFilter, (void**)&gcap.pVCap);
            pM->Release();                            // We don't need the monike
        break;
      }
          pM->Release();
      uIndex++;
    }
    pEm->Release();                             // We've got the device; don't need
```

After AMCap has a device, it retrieves the interface pointers to measure frames, get the driver name, and get the capture size. AMCap stores each pointer in the gcap global structure.

```
    // We use this interface to get the number of captured and dropped frames
    gcap.pBuilder->FindCaptureInterface(gcap.pVCap,
                              IID_IAMDroppedFrames, (void **)&gcap.pDF);

    // We use this interface to get the name of the driver
    gcap.pBuilder->FindCaptureInterface(gcap.pVCap,
                              IID_IAMVideoCompression, (void **)&gcap.pVC);
    // We use this interface to set the frame rate and get the capture size
    gcap.pBuilder->FindCaptureInterface(gcap.pVCap,
                                IID_IAMVideoStreamConfig, (void **)&gcap.pVSC);
```

AMCap then gets the media type and sizes the display window to match the size of the video format.

```
  AM_MEDIA_TYPE *pmt;
    gcap.pVSC->GetFormat(&pmt);                    // Current capture format

    ResizeWindow(HEADER(pmt->pbFormat)->biWidth,
                                HEADER(pmt->pbFormat)->biHeight);
    DeleteMediaType(pmt);
```

This section applies only to earlier Video for Windows devices. Video for Windows devices support a specific set of dialog boxes, which set the video source, format, and display type. For additional information on these dialog boxes, see the IAMVfwCaptureDialogs interface documentation.

```
    hr = gcap.pBuilder->FindCaptureInterface(gcap.pVCap,
                                IID_IAMVfwCaptureDialogs, (void **)&gcap.pDlg);
    if (hr != NOERROR) {
        ErrMsg("Error %x: Cannot find VCapture:IAMVfwCaptureDialogs", hr);
    }
```

Now that AMCap has the video capture device and its relevant information, it repeats the process with the audio devices and stores the information in the global structure. Note that it calls ICreateDevEnum::CreateClassEnumerator with the CLSID_AudioInputDeviceCategory CLSID to enumerate audio hardware devices.

```
    hr = CoCreateInstance(CLSID_SystemDeviceEnum, NULL, CLSCTX_INPROC_SERVER,
                          IID_ICreateDevEnum, (void**)&pCreateDevEnum);
    uIndex = 0;
    hr = pCreateDevEnum->CreateClassEnumerator(CLSID_AudioInputDeviceCategory,
                                                                  &pEm, 0);
    pCreateDevEnum->Release();

    pEm->Reset();
    gcap.pACap = NULL;
    while(hr = pEm->Next(1, &pM, &cFetched), hr==S_OK)
    {
        if ((int)uIndex == gcap.iAudioDevice) {                  // this is the one
            hr = pM->BindToObject(0, 0, IID_IBaseFilter, (void**)&gcap.pACap);
                    pM->Release();
            break;
        }
                pM->Release();
        uIndex++;
    }
    pEm->Release();
```

AMCap also repeats the process of retrieving the format interface, this time for the audio device.

```
    hr = gcap.pBuilder->FindCaptureInterface(gcap.pACap,
                                IID_IAMAudioStreamConfig, (void **)&gcap.pASC);
}
```

How to Store DirectShow Filter Properties Persistently

The Win32 IPropertyBag and IPersistPropertyBag interfaces store and retrieve groups ("bags") of properties for developer-specified objects. Properties stored by these interfaces are persistent; that is, they remain consistent between different instantiations of the same object. Filters can store their properties (CLSID, FriendlyName, and DevicePath) persistently. After a filter stores its properties, DirectShow automatically retrieves them whenever it instantiates the filter. To add this functionality to your filter, implement the **IPersistPropertyBag** interface and its Load method. Your implementation of the **Load** method should call the IPropertyBag::Read method to load the filter's properties into a Win32 VARIANT variable, and then initialize its input and output pins.

The following code sample demonstrates how the DirectShow VfWCapture filter implements the IPersistPropertyBag::Load method. Remember that your filter must supply a valid IPropertyBag pointer to hold the filter's properties during execution. You can specify an error log to trap errors generated by the filter's properties, although you can pass in a null value to ignore error reporting.

```
STDMETHODIMP CVfwCapture::Load(LPPROPERTYBAG pPropBag, LPERRORLOG pErrorLog)
{
    HRESULT hr;
    CAutoLock cObjectLock(m_pLock);              // Locks the object; automatically

    if (m_pStream)                               // If the filter already exists fo
        return E_UNEXPECTED;

    VARIANT var;                                 // VARIANT from Platform SDK
    var.vt = VT_I4;                              // four-byte integer (long)
    hr = pPropBag->Read(L"VFWIndex", &var, 0);   // VFWIndex is the private name us
    if(SUCCEEDED(hr))                            // If it read the properties succe
    {
        hr = S_OK;                               // Defaults return value to S_OK
        m_iVideoId = var.lVal;                   // Stores the specified hardware d
        CreatePins(&hr);                                 // Inits the pins, replaci
    }
    return hr;                                   // Returns S_OK or an error value,
}
```

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

Clocks

This section describes time and synchronization in DirectShow, how to implement a reference clock in a filter or application, and how to make a reference clock the master clock if a filter graph has more than one clock.

Previous Home Topic Contents Index Next

Synchronization

DirectShow accomplishes synchronization by using a reference clock. A reference clock is an object that implements the IReferenceClock interface. For example, because sound cards are often used for reference clocks, the audio renderer filter implements this interface, which essentially allows any caller to register for the receipt of time notifications.

Previous Home Topic Contents Index Next

Understanding Time and Clocks in DirectShow

This article describes the basic concepts of time used in the filter graph and then goes on to describe what a reference clock is, how it is implemented by a filter or as a stand-alone clock, how the filter graph manager decides which clock to use as the master reference clock, and how to ensure that a reference clock implemented by a filter is used as the master reference clock.

Contents of this article:

About Time

A few concepts of time come up often in discussions about DirectShow streams, synchronization to a common clock, and seeking to different places in the stream. Four terms are defined here:

- Media time

- Reference time
- Stream time
- Presentation time

In DirectShow, the term *media time* is used to refer to positions within a seekable medium such as a file on disk. Media time can be expressed in a variety of units, such as frames, seconds, bytes, or 100-nanosecond intervals, and indicates a position within the data in the file.

Reference time is an absolute time (sometimes called wall-clock time) that is established by a reference clock in the filter graph. It is a reference to some time value outside the filter graph (for example, perhaps the number of milliseconds since Windows was started).

Stream time is relevant only within a running filter graph, and represents the time since the graph was last started. When a filter graph is run, each filter is passed a notional start time (tStart) according to the reference clock, and the packets of data that a filter receives will normally be time-stamped with the stream time at which they should be presented. This is known as the *presentation time*. Stream time is often called "relative reference time" since, by definition, stream time is equivalent to reference time minus start time when the graph is running.

Since a filter graph can start playing a file at an arbitrary position and rate, file source filters and/or parsers must take these two factors into account when time-stamping the samples that they pass downstream to renderers. Such filters will calculate the presentation time and will place that value in the sample. The presentation time is calculated by subtracting the starting media time (the last time that was seeked to) from the media time of the sample, and dividing this by the playback rate. Expressed as a formula, this would be:

```
Presentation Time = (Media Time - Starting Media Time)  / playback rate.
```

For example, consider a media stream with a duration of six seconds that is set to be played at double speed. What happens when the filter graph is seeked to a sample with a media time of two seconds and then run? Each media sample read from the disk gets stamped with a presentation time equivalent to half of the difference of its media time and the start time (two seconds). Here is how the time stamps would appear at one-second media sample intervals:

Media time (sec)	**Presentation time stamp (sec)**
3	0.5
4	1.0
5	1.5
6	2.0

When finally presented in the renderer, the difference between the actual time the sample is rendered and the stamped presentation time that was expected can be calculated. In a perfect graph, this would always be zero. In reality, there is a margin of acceptable tolerance. If this difference is out of tolerance, then quality-control management will be initiated by the renderer.

About Reference Clocks

A *reference clock* is an object that implements the IReferenceClock interface. This interface supports querying for the current time and scheduling events according to time as counted by that clock. Event scheduling is achieved by submitting advise requests to the clock. These requests can be for single-shot or periodic events.

Many pieces of hardware can provide time signals. These time signals can be of particularly high accuracy, or might represent some clock signal significant only to the resolution of a particular application, such as sound playback.

Filters can expose a hardware time signal to other filters by implementing a reference clock in the filter graph. A filter graph manager will choose (or be assigned) one of these reference clocks to be the *filter graph reference clock*. (By definition, there is only one reference clock allowed in a filter graph.) If no such reference clocks exist, the filter graph manager can create a suitable reference clock and use that one instead. A reference clock can be appointed by calling the filter graph manager's IMediaFilter::SetSyncSource method. The reference clock is also called the *sync source*. A filter graph manager propagates this selection to the filters in its graph by calling their individual **IMediaFilter::SetSyncSource** methods.

Developers can provide a reference clock on a filter for purely altruistic reasons; the filter might simply be in a position to provide a high-accuracy clock. Alternatively, the overall performance of a filter graph might be determined by which reference clock, of all the possible reference clocks in the graph, is selected to provide its services to the filter graph. Because audio hardware cannot easily adjust the rate at which it delivers data, it is often the most appropriate source of time signals. Therefore, the reference clock of the audio renderer is often selected to be the filter graph's reference clock.

All clocks in DirectShow report a reference time; that is, a time which would be suitable to use for the filter graph reference time. The filter graph reference time for the filter graph is the time of the clock that has been selected as the current sync source.

Characteristics of a Reference Clock

Any reference clock must support the IReferenceClock interface. The time of the clock can be obtained by calling the IReferenceClock::GetTime method. The time returned by GetTime is defined as a REFERENCE_TIME type (LONGLONG) and loosely represents the number of 100-nanosecond units that have elapsed since some fixed start time. This is just a guideline. Specifically, **IReferenceClock::GetTime** must adhere to some conditions as follows.

A reference clock must return values that are monotonically increasing. That is, successive calls to GetTime must result in values that are greater than or equal to the previous value.

Also, the return value should generally increase at a rate of approximately one per 100 nanoseconds.

In exceptional circumstances, it is allowable for the clock to stop for a time. (This will effectively suspend any filter that was using the clock as a sync source.) Furthermore, it is allowable for the clock to jump forward in exceptional circumstances.

Finally, the reference clock must continue to count time even if its containing filter graph is stopped, and should normally continue to count time if it is paused. (A filter's reference clock implementation can optionally use a system-supplied clock to fill in during such times, but that is an implementation decision.)

The reference clock does not have to bear any permanent relationship to any real time. It is allowed to drift, it can drift at a changing rate, and it need not correct for such drift. In particular, it does not have to represent a count of the number of 100 nanoseconds that have passed since some arbitrary time in the past. It is important to remember that this loose description of a reference clock, though it can be helpful, is just a guideline. In some cases, a strict adherence to the guideline might actually result in a poorer overall look and feel when the filter graph is running. If you want your clock to adhere strictly to the guideline, you need to set the clock yourself.

Using a Reference Clock

A filter will always be told to use a specific clock (or, possibly, to use none) by a call to its IMediaFilter::SetSyncSource method. Filters that require timing information should use the clock that they are told to use. All filters in a particular filter graph should use the same reference clock. An application can use a reference clock by calling IMediaFilter::GetSyncSource on the filter graph manager to obtain a pointer to an IReferenceClock, and then invoke methods on that interface. If a null pointer is passed to SetSyncSource, it implies that the filter should not use any clock and should just run as quickly as possible without discarding any data. If no clock has been set as the reference clock for the filter graph, then when the filter graph manager's GetSyncSource is called, the filter graph manager chooses a clock in the filter graph or creates and appoints a clock of its own. This is the same logic that applies when the filter graph is first run.

If a new reference clock is appointed, the time as tracked by the old reference clock and the time as tracked by the new reference clock need bear no relation to each other. As a consequence, functions that call IReferenceClock::GetTime on the current sync source should not be surprised to see the reported time jump forward or backward. Reference clocks can be switched only if the filter graph is paused or stopped. When the filter graph next starts to run, the filters in the filter graph will be given their start times in terms of the new clock. (See IMediaFilter::Run for details.) Typically, only filters that use advise requests from the reference clock (that is, use its scheduling facilities) must specifically handle clock differences when then the filter graph is switched to an alternative sync source.

If a filter (or application) uses a reference clock's scheduling facilities, it is important to recognize that the advise requests are scheduled against that specific clock in the absolute time used by that clock. If a filter has set up advise requests against its sync source, and is then notified of a new sync source, then the filter is normally expected to cancel the advise requests on the first clock and set them up again on the new one. Applications that use advise requests should monitor for EC_CLOCK_CHANGED events. If an **EC_CLOCK_CHANGED** event notification is received, then the application should cancel any outstanding advise requests, call GetSyncSource on the filter graph manager to obtain an interface pointer to the new clock, and reschedule the advise requests on the new clock (also taking into account that the time on the old and new clock might be different).

Similarly, when a filter sets up advise requests in stream time (for example, 135 milliseconds into the media stream), then it is expected that the filter will set up an advise when it is told to run, cancel the advise if it is told to pause or stop, and recalculate and resubmit the advise request when it is told to run again.

DirectShow Clock Classes

DirectShow provides three class that are used to implement clocks:

- CBaseReferenceClock, the main clock class that implements IReferenceClock.
- CAMSchedule, which handles the mechanics of advise list processing and is inherited by CBaseReferenceClock.
- CSystemClock, a stand-alone minimal clock class derived from CBaseReferenceClock.

CBaseReferenceClock provides the event notification functionality (mainly via CAMSchedule) and a rudimentary clock based on the Win32 timeGetTime function.

The most important aspect of CBaseReferenceClock is a virtual GetPrivateTime method. This method can be overridden in derived classes to return a time. The CBaseReferenceClock::GetTime method calls **GetPrivateTime**, caches the result, and ensures that the time it returns to its caller does not go backward. Thus, implementers of **GetPrivateTime** can code that method so that it returns a best estimate, and not worry about time going backward. **CBaseReferenceClock::GetTime** locks the clock before calling GetPrivateTime; therefore, implementations of **GetPrivateTime** need not worry about locking the clock. If methods in derived classes call **GetPrivateTime**, they should ensure that the clock is locked first and released afterward.

A derived clock can basically be implemented in one of two ways:

- It can override GetPrivateTime (and SetTimeDelta if desired) and provide its own clock. This effectively abandons the clock in CBaseReferenceClock.
- It can call SetTimeDelta from the derived clock to minimally adjust the time of the clock in CBaseReferenceClock.

CSystemClock is derived from CBaseReferenceClock and implements a stand-alone clock (not attached to a filter), which can be saved as part of a stored filter graph and used as the filter graph reference clock when the filter is restored. **CSystemClock** generates the default time base generated by **CBaseReferenceClock** (using the Win32 timeGetTime function).

Multiple Clocks in a Filter Graph

It sometimes happens that a filter graph will be built with more than one clock. Several filters in the graph might implement clocks or there might even be an independent system clock in the filter graph. Since only one clock can be the master clock, it is assumed that all other clocks, when notified of the sync source, will synchronize with it.

The filter graph manager has a default algorithm for choosing the master reference clock, and a filter uses this to ensure that its own reference clock becomes the master clock. Why would a filter want to insist on its own reference clock rather than letting the filter graph manager make the decision? There are several reasons to use a filter's own reference clock. For example, the filter's clock might:

- Be tied to some external source that the filter graph must be synchronized with.
- Be the most accurate.
- Incur the lowest system overhead while being used.
- Be the only clock that cannot be adjusted to be in sync with the other(s). (Although, it could be argued that this constitutes a badly designed clock.)

Here are the steps used by the filter graph manager for choosing the master reference clock in a filter graph:

1. If a call to the filter graph manager's IMediaFilter::SetSyncSource method has been made, then that reference clock will be used (or no reference clock will be used if a null pointer was passed to **IMediaFilter::SetSyncSource**).
2. If IMediaFilter::SetSyncSource has never been called for this graph, the sync source is provided by the first connected filter that exposes the IReferenceClock interface. In this

case, the search for the first connected filter goes in roughly upstream order, starting with the renderers. Connected means the filter has an input pin connected to another filter. There is no check to see if that stream would actually be active. If more than one clock is found at the same level in this search, and both are connected, it is undefined which one will be used as the sync source for this filter graph. The filter graph manager will choose one of them.
3. If neither of those steps result in a sync source being set, the filter graph manager will create a freestanding reference clock and use that as the sync source.

A filter can explicitly indicate which reference clock is to be the sync source by having the filter's IBaseFilter::JoinFilterGraph method call IMediaFilter::SetSyncSource on the filter graph manager when it joins the filter graph to set the desired clock. In fact, if the filter really needs its clock to be the reference clock, to the extent that the filter won't function properly if it isn't, then it should additionally fail the **IBaseFilter::JoinFilterGraph** call if the **IMediaFilter::SetSyncSource** call fails.

Having described how to force a filter's clock to be the system clock, it should be emphasized that this is not normally required.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

Controlling Filter Graphs Using C

This article describes how to use the interfaces and methods exposed by the Microsoft® DirectShow™ dynamic-link library to communicate with the Filter Graph Manager and the filters in a graph. These interfaces and methods render a stream of time-stamped video data in applications that are based in Microsoft Windows®. This article provides an overview of the interfaces and methods to use, and then describes their use in the DirectShow CPlay sample application.

Contents of this article:

Interfaces that Access the Filter Graph Manager

The stream architecture enables applications to communicate with the filter graph manager, and the filter graph manager to communicate with individual filters to control the movement of data through the filter graph. It also enables filters to post events that an application can retrieve, so an application can, for example, retrieve status information about a special filter it has installed.

This section contains the following topics.

- Implementing Dual Interfaces
- Installing and Registering Quartz.dll
- Instantiating the Filter Graph Manager
- Invoking Methods on the Interfaces

An application communicates with the filter graph manager and the filters in a specific graph by using the interfaces exposed by either the filter graph manager or the filters. The following table identifies these interfaces.

Interface	**Description**
IAMCollection	Represents a collection of objects of type IFilterInfo, IRegFilterInfo, IMediaTypeInfo, and IPinInfo.
IBasicAudio	Controls and retrieves current volume setting.
IBasicVideo	Controls a generic video renderer.
IDeferredCommand	Used in conjunction with IQueueCommand methods to defer the execution of methods and properties.
IFilterInfo	Enables an Automation client to set and retrieve filter properties.
IGraphBuilder	Builds the filter graph manager.
IMediaControl	Instantiates the filter graph and controls media flow (runs, pauses, stops).
IMediaEvent	Enables customized event handling for events such as repainting, user termination, completion, and so on.
IMediaPosition and IMediaSeeking	Controls or retrieves start time, stop time, preroll rate, and current position.
IMediaTypeInfo	Enables an Automation client to retrieve a media type's major type and subtype.
IQueueCommand	Enables an application to queue methods and properties so that the filter invokes them during rendering of a video stream.
IPinInfo	Enables an Automation client to set and retrieve filter properties.
IRegFilterInfo	Enables an Automation client to retrieve the name of a registered filter and add a filter to the filter graph.
IVideoWindow	Controls window aspects of a video renderer, such as the window's position and size.

Of all the interfaces for the filter graph manager, C and C++ programmers use the following most effectively.

- IBasicAudio
- IBasicVideo
- IDeferredCommand
- IGraphBuilder
- IMediaControl
- IMediaEvent
- IMediaSeeking
- IQueueCommand
- IVideoWindow

The remainder are collection interfaces, which enable Automation clients, such as Microsoft Visual Basic®, to access the properties of filters, pins, and media types that are not otherwise exposed to Automation clients.

Implementing Dual Interfaces

Most of the interfaces that communicate with the Filter Graph Manager are implemented as dual interfaces. This means that an application can call the methods in each interface directly or through Automation (by using the IDispatch::Invoke method). DirectShow provides Automation support for the developer using Visual Basic. The developer using C or C++ can avoid the indirection (and accompanying overhead) associated with Automation by calling the methods directly.

DirectShow doesn't implement all interfaces as dual interfaces. An application must call the methods in these interfaces directly. For example, the following interfaces are not dual interfaces: IQueueCommand, IDeferredCommand, and IGraphBuilder.

Installing and Registering Quartz.dll

Before you begin using the filter graph manager, you must install and register the Quartz.dll dynamic-link library. Currently, the DirectShow SDK setup program automates this process. Run Setup.exe and choose the Runtime option. This program copies Quartz.dll to your Windows\System directory and adds the appropriate entries to your system's registration database.

Instantiating the Filter Graph Manager

After you have registered Quartz.dll, you can begin using the filter graph manager in your Windows-based application. First, initialize the COM library by calling the COM CoInitialize function. The sample application calls **CoInitialize** within its InitApplication function in the Cplay.c file of the CPlay sample application.

Next, instantiate the filter graph manager. Most applications should use the CoCreateInstance function to instantiate the filter graph. Both **CoCreateInstance** and CoGetClassObject can instantiate an object; however, applications typically use the former to instantiate a single object and the latter to instantiate multiple instances of an object.

The complete call to CoCreateInstance appears as follows:

```
hr = CoCreateInstance( &CLSID_FilterGraph,          // Get this document's graph ob
                        NULL,
                        CLSCTX_INPROC_SERVER,
                        & IID_IGraphBuilder,
                        (void **) &media.pGraph);
```

The first parameter, CLSID_FilterGraph, is the class identifier (CLSID) for the filter graph manager. This CLSID is defined in the Uuids.h file, which is installed as part of the DirectShow SDK. The CLSID is a 128-bit value that the registration database uses to identify the dynamic-link library (DLL or in-process server). Using this value, COM can locate and then load the appropriate DLL.

The second parameter is a pointer to the outer IUnknown and is NULL because the Filter Graph object is not part of an aggregate.

The third parameter is the context in which the code that manages the Filter Graph will run, which is in the same process as the caller of the CoCreateInstance function.

The fourth parameter passed to the CoCreateInstance function identifies the interface that the application will use to communicate with the object. This interface identifier should be IID_IGraphBuilder; this value is defined internally in the DirectShow sources and then exposed in the Strmif.h file.

If the call to CoCreateInstance succeeds, this function returns a pointer to a filter graph manager object in the media.pGraph variable. After this pointer is returned, the application begins to call the methods in the IGraphBuilder interface. Typically, the application first calls the IGraphBuilder::RenderFile method. This method creates a filter graph for the type of file that was supplied as one of the parameters. In addition, the application can use the IGraphBuilder::QueryInterface method to retrieve pointers to any of the interfaces exposed by the filter graph manager. The **IGraphBuilder** interface derives from IUnknown.

If you are writing your application in C (rather than C++), you must use a vtable pointer to call the methods exposed by IGraphBuilder. The following example illustrates a call to the QueryInterface method on the **IGraphBuilder** interface from within an application written in C.

```
hr = media.pGraph->lpVtbl->QueryInterface(media.pGraph,
    &IID_IMediaEvent, (void **) &pME);
```

If you are writing your application in C++, the function is simpler; it requires less indirection and one less parameter:

```
hr = m_pGraph->QueryInterface(IID_IMediaEvent, (void **) &pME);
```

Invoking Methods on the Interfaces

An application can retrieve a pointer to any of the other interfaces exposed by the filter graph manager by calling the IGraphBuilder::QueryInterface method and supplying a REFIID for the corresponding interface. After retrieving this interface pointer, the application can begin calling the interface's methods by using the interface's vtable pointer (just as the IGraphBuilder's vtable pointer called the **IGraphBuilder::QueryInterface** method in the previous example). The application must release an acquired interface by calling the IUnknown::Release method on that interface.

CPlay Tutorial

This section's tutorial describes CPlay, a sample included in the DirectShow SDK that plays a media file. The source files for this application are in the Samples\DS\Player\CPlay subdirectory of the DirectShow SDK project.

This section contains the following topics.

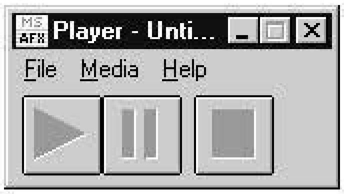
- CPlay Sample Application
- Files in CPlay

This tutorial does not describe the Microsoft Windows® API code found in the source files. Instead, it focuses almost exclusively on the code that shows:

- How to instantiate a filter graph for a particular file type.
- How to process media events.
- How to run, pause, and stop the media stream.
- How to set a global variable to indicate the valid media state (running, paused, or stopped).
- How to release the resources and clean up the variables used by the filter graph.

CPlay Sample Application

You can use the CPlay sample application to open a media file and then run, pause, or stop the corresponding media stream. The application's user interface consists of menus and a toolbar. The menus include File, Media, and Help. The toolbar includes Play, Pause, and Stop buttons.



After you open a file and click Play, the filter graph renders the video stream in its default window.

Files in CPlay

The sample application consists of six source files. Each file contains source code that accomplishes a specific set of tasks. For example, the About.c module contains the code that displays the About dialog box. The following table identifies each source file and describes its purpose.

File	Description
About.c	Displays the About dialog box.
Assert.c	Displays a message box with debugging information.
Cplay.c	Processes user input.
File.c	Displays the File Open dialog box.
Media.c	Instantiates the filter graph; invokes the filter graph methods to run, pause, and stop the video rendering.
Toolbar.c	Draws the toolbar buttons.

The remainder of this article focuses primarily on the code found in the Media.c file; however, references to other files appear when describing some of the tasks accomplished by this application.

Using the Filter Graph Manager

The Media.c file contains initialization, destruction and cleanup, command handling, and state change code. The initialization code instantiates a filter graph for a particular file type. The destruction code releases the resources and cleans up the variables used by the filter graph. The command handling code invokes the methods required to play, pause, or stop the video rendering. The state change code sets a global variable that indicates valid media states (that is, can stop, can pause, can play).

This section contains the following topics.

- Initializing the Filter Graph Manager and the Filter Graph
- Playing, Pausing, and Stopping the Video Stream
- Handling Events

Initializing the Filter Graph Manager and the Filter Graph

The following code illustrates how to create the filter graph manager and the filter graph, including including how to enable event handling, and how to open the media file that the filter graph will render.

First, instantiate the filter graph manager. The CreateFilterGraph function in Media.c instantiates the filter graph manager by calling the COM CoCreateInstance function. It saves the pointer returned by **CoCreateInstance** in the pGraph member of a global media structure (defined in Media.h in the CPlay sample included in the SDK).

```
BOOL CreateFilterGraph()
{
    HRESULT hr;

    hr = CoCreateInstance(&CLSID_FilterGraph,  // CLSID of object
           NULL,                             // Outer unknown.
           CLSCTX_INPROC_SERVER,             // Type of server.
           &IID_IGraphBuilder,               // Interface wanted.
               (void **) &media.pGraph);         // Pointer to IGraphBuilder.
...
}
```

Next, enable event handling. Using the pointer returned by CoCreateInstance, the CreateFilterGraph function retrieves a pointer to the IMediaEvent interface by calling the IUnknown::QueryInterface method. The interface pointer retrieves an event notification handle by calling the IMediaEvent::GetEventHandle method. The main message loop uses this handle (the DoMainLoop function in CPlay.c). After GetEventHandle obtains the handle, CreateFilterGraph releases the pointer to the **IMediaEvent** interface by calling the IUnknown::Release method.

```
IMediaEvent *pME;

hr = media.pGraph->lpVtbl->QueryInterface(media.pGraph, &IID_IMediaEvent, (void **)
    if (FAILED(hr)) {
          DeleteContents(); //Releases the pointer media.pGraph.
          return FALSE;
    }

    hr = pME->lpVtbl->GetEventHandle(pME, (OAEVENT*) &media.hGraphNotifyEvent);
    pME->lpVtbl->Release( pME );
```

After instantiating the Filter Graph Manager and enabling event handling, open the media file to be rendered. In the CPlay sample application, a user opens a multimedia file. The file name extension (for example, .avi or .mpg) is unimportant, because the DirectShow filter graph examines the file header to ensure that the file is a multimedia file.

When the user opens a file by choosing Open from the File menu, this action calls the OpenMediaFile function in Media.c, which displays the File Open common dialog box.

```
 void OpenMediaFile( HWND hwnd, LPSTR szFile ){
// File..Open has been selected
    static char szFileName[ _MAX_PATH ];
    static char szTitleName[ _MAX_FNAME + _MAX_EXT ];
    // The user has already chosen a file.
    if( szFile!=NULL && RenderFile( szFile ) ){
        LPSTR szTitle;

        // Work out the full path name and the file name from the
        // specified file.
        GetFullPathName( szFile, _MAX_PATH, szFileName, &szTitle );
        strncpy( szTitleName, szTitle, _MAX_FNAME + _MAX_EXT );
        szTitleName[ _MAX_FNAME + _MAX_EXT -1 ] = '\0';

        // Set the main window title and update the state.
        SetTitle( hwnd, szTitleName );
        ChangeStateTo( Stopped );
// The user hasn't already chosen a file, so display the Open File
// dialog box. The DoFileOpenDialog function is in file.c in the CPlay
// sample.
    } else if( DoFileOpenDialog( hwnd, szFileName, szTitleName )
               && RenderFile( szFileName ) ){

        // Set the main window title and update the state.
        SetTitle( hwnd, szTitleName );
        ChangeStateTo( Stopped );
    }
}
```

After the file has been opened, render the file. The OpenMediaFile function passes the name of the user's chosen file to the RenderFile function in Media.c. The RenderFile function in turn calls the CreateFilterGraph function to instantiate the filter graph manager. After creating the filter graph manager, the RenderFile function calls the IGraphBuilder::RenderFile method to create the actual filter graph:

```
BOOL RenderFile( LPSTR szFileName )
{
    HRESULT hr;
    WCHAR wPath[MAX_PATH];
    DeleteContents(); // Release the pointer media.pGraph if it exists,
                      // because the call to CreateFilterGraph will
                      // retrieve a new pointer.

   //Create the filter graph manager
    if ( !CreateFilterGraph() ) {
          PlayerMessageBox( IDS_CANT_INIT_QUARTZ );
          return FALSE;
    }

    MultiByteToWideChar( CP_ACP, 0, szFileName, -1, wPath, MAX_PATH );
    SetCursor( LoadCursor( NULL, IDC_WAIT ) ); // Put up the hour-glass
                                               // while the media file
                                               // loads.
    // Create the actual filter graph
    hr = media.pGraph->lpVtbl->RenderFile(media.pGraph, wPath, NULL);
    SetCursor( LoadCursor( NULL, IDC_ARROW ) ); // Turn the cursor back
```

```
                                                        // to an arrow.
    if (FAILED( hr )) {
          PlayerMessageBox( IDS_CANT_RENDER_FILE );
          return FALSE;
    }
    return TRUE;

}
```

Playing, Pausing, and Stopping the Video Stream

After the application creates the filter graph manager and the filter graph, it can expose the user interface, which enables the user to play, pause, and stop video rendering. In the case of CPlay, the toolbar buttons (Play, Pause, and Stop) are redrawn in color after the user chooses a valid file.

When the user clicks Play, the OnMediaPlay function is called. This function accomplishes the following tasks sequentially.

1. Examines the global state variable in the media structure to ensure that the video can be rendered.
2. Retrieves a pointer to the IMediaControl interface.
3. Invokes the IMediaControl::Run method.
4. Releases the IMediaControl interface.
5. Sets the global state variable.

The OnMediaPlay function appears as follows:

```
void OnMediaPlay( void ){
    if( CanPlay() ){
        HRESULT   hr;
        IMediaControl *pMC;

        // Obtain the interface to our filter graph.
        hr = media.pGraph->lpVtbl->QueryInterface(media.pGraph,
            &IID_IMediaControl, (void **) &pMC);

        if( SUCCEEDED(hr) ){
            // Ask the filter graph to play and release the interface.
            hr = pMC->lpVtbl->Run( pMC );
            pMC->lpVtbl->Release( pMC );

            if( SUCCEEDED(hr) ){
                ChangeStateTo( Playing );
                return;
            }
        }

        // Inform the user that an error occurred.
        PlayerMessageBox( IDS_CANT_PLAY );

    }
}
```

The code that handles pausing and stopping the video stream is nearly identical to the code that plays the media stream. The actual functions that handle these tasks are OnMediaPause and OnMediaStop, respectively. You can find all this code in the Media.c file.

Handling Events

The IMediaEvent interface enables an application to receive events that the filter graph or individual filters within the graph raise. Following are some of the possible events and corresponding event notification messages.

Event notification message	**Description**
EC_COMPLETE	The video has finished rendering.
EC_USERABORT	A user forced the termination of a requested operation.
EC_ERRORABORT	An error forced the termination of a requested operation.
EC_PALETTE_CHANGED	The video palette changed.
EC_REPAINT	The display should be repainted.

The sample application tracks the EC_COMPLETE, EC_USERABORT, and EC_ERRORABORT events by using the IMediaEvent::GetEvent method. The application calls this method from within the OnGraphNotify function. The application calls the OnGraphNotify function (in Media.c) from within the application's main message loop function (DoMainLoop), which you can find in the Cplay.c file.

If any of these events are raised, OnGraphNotify immediately stops video rendering by calling the OnMediaStop function.

The OnGraphNotify function accomplishes the following tasks sequentially.

1. Declares the IMediaEvent interface pointer and the variables for the event code and event parameters.
2. Retrieves a pointer to the IMediaEvent interface by calling IUnknown::QueryInterface.
3. Calls the IMediaEvent::GetEvent method to retrieve the next event notification. The retrieved event is stored in the lEventCode variable and the event parameters are stored in the lParam1 and lParam2 variables. The time-out value is set to zero, which means that GetEvent will not wait for an event to occur, but only return an already waiting event.
4. Checks the event type stored in lEventCode and takes the appropriate action, if GetEvent retrieves an event. See Event Notification Codes for a list of the system-supplied events that DirectShow supports. Note that if the event parameters are declared as type **BSTR** instead of **LONG**, IMediaEvent::FreeEventParams should be called free the **BSTR**s.

```
void OnGraphNotify(void) {
    IMediaEvent *pME;

    long lEventCode, lParam1, lParam2;

    ASSERT( media.hGraphNotifyEvent != NULL );

    if( SUCCEEDED(media.pGraph->lpVtbl->QueryInterface(media.pGraph,
            &IID_IMediaEvent, (void **) &pME))){
        if( SUCCEEDED(pME->lpVtbl->GetEvent(pME, &lEventCode, &lParam1,
                &lParam2, 0))
            && ( lEventCode == EC_COMPLETE
                || lEventCode == EC_USERABORT
                || lEventCode == EC_ERRORABORT
                )
          )
            OnMediaStop();
              pME->lpVtbl->Release( pME );
              }
}
```

Previous Home Topic Contents Index Next

Creating a Capture Application

Microsoft® DirectShow™ provides the capability to capture and preview both video and audio data from an application, when combined with the appropriate capture hardware. The data source might include a VCR, camera, TV tuner, microphone, or other source. An application can display the captured data immediately (preview) or save it to a file for later viewing either inside or outside of the application.

DirectShow takes advantage of new capture drivers that are written as DirectShow filters, and also uses existing Video for Windows-style drivers.

Note This article relies heavily on the AMCap Sample (DirectShow Capture Application) sample application. See the AMCap sample code (Amcap.cpp) in the Samples\DS\Capture directory of the DirectShow SDK for complete sample code, because this article does not present AMCap Sample (DirectShow Capture Application) in its entirety.

The AMCap Sample (DirectShow Capture Application) sample application performs video and audio capture, similar to the VidCap sample from Video for Windows®. It uses the ICaptureGraphBuilder interface to handle the majority of the capture work. In your own capture application, you'll use the same methods and interfaces that AMCap uses. This article focuses on AMCap's use of **ICaptureGraphBuilder** to perform video and audio capture and presents relevant code excerpts from AMCap.

This article assumes you are familiar with the DirectShow filter graph architecture and the general layout of a capture filter graph. See Filter Graph Manager and Filter Graphs and About Capture Filter Graphs for more information.

Contents of this article:

Introduction to ICaptureGraphBuilder

The ICaptureGraphBuilder interface provides a filter graph builder object that applications use to handle some of the more tedious tasks involved in building a capture filter graph, which frees the application to focus on capture. You access the graph builder object by calling methods on **ICaptureGraphBuilder**. The variety of methods satisfies the basic requirements for capture and preview functionality.

The FindInterface method searches for a particular capture-related interface in the filter graph. The method handles the complexities of filter graph traversal for you, which enables you to access the functionality of a particular interface without having to enumerate pins and filters in the filter graph looking for the interface. The RenderStream method connects source filters to rendering filters, optionally adding other needed intermediate filters. The ControlStream method independently control sections of the graph for frame-accurate start and stop.

Additional methods deal with allocating space for the capture file (AllocCapFile), specifying a name for it and building up the *file writer* section of the graph, which consists of the multiplexer and file writer filters (SetOutputFileName), and saving the captured data to another file (CopyCaptureFile). Finally, SetFiltergraph and GetFiltergraph enable the application to provide a filter graph for the graph builder to use or retrieve the filter graph already in use.

Device Enumeration and Capture Interfaces

AMCap's InitCapFilters function enumerates the capture devices on the system by using the ICreateDevEnum::CreateClassEnumerator method. After enumerating a capture device and instantiating a DirectShow filter to use that device, the sample calls the ICaptureGraphBuilder::FindInterface method several times to obtain interface pointers for the IAMDroppedFrames, IAMVideoCompression, IAMStreamConfig, and IAMVfwCaptureDialogs capture-related interfaces. The AMCap code saves all of these interface pointers for later use in the gcap global structure and uses gcap structure members throughout the code.

Note: IAMVfwCaptureDialogs is designed for you to use only with the Microsoft-supplied video capture filter and only when using a former Video for Windows device.

For your convenience, the declaration of the gcap structure follows:

```
struct _capstuff {
    char szCaptureFile[_MAX_PATH];
    WORD wCapFileSize;  // size in Meg
    ICaptureGraphBuilder *pBuilder;
    IVideoWindow *pVW;
    IMediaEventEx *pME;
    IAMDroppedFrames *pDF;
    IAMVideoCompression *pVC;
    IAMVfwCaptureDialogs *pDlg;
    IAMStreamConfig *pASC;      // for audio cap
    IAMStreamConfig *pVSC;      // for video cap
    IBaseFilter *pRender;
    IBaseFilter *pVCap, *pACap;
    IGraphBuilder *pFg;
    IFileSinkFilter *pSink;
    IConfigAviMux *pConfigAviMux;
    int  iMasterStream;
    BOOL fCaptureGraphBuilt;
    BOOL fPreviewGraphBuilt;
    BOOL fCapturing;
    BOOL fPreviewing;
    BOOL fCapAudio;
    int  iVideoDevice;
    int  iAudioDevice;
    double FrameRate;
    BOOL fWantPreview;
    long lCapStartTime;
    long lCapStopTime;
    char achFriendlyName[120];
    BOOL fUseTimeLimit;
    DWORD dwTimeLimit;
} gcap;
```

AMCap's InitCapFilters function stores several interface pointers in the gcap structure. Be sure to properly release all interface pointers when they are no longer needed as shown in the following example.

```
    if (gcap.pBuilder)
      gcap.pBuilder->Release();
    gcap.pBuilder = NULL;
    if (gcap.pSink)
      gcap.pSink->Release();
    gcap.pSink = NULL;
    if (gcap.pConfigAviMux)
      gcap.pConfigAviMux->Release();
    gcap.pConfigAviMux = NULL;
    if (gcap.pRender)
      gcap.pRender->Release();
    gcap.pRender = NULL;
    if (gcap.pVW)
      gcap.pVW->Release();
    gcap.pVW = NULL;
    if (gcap.pME)
      gcap.pME->Release();
    gcap.pME = NULL;
    if (gcap.pFg)
      gcap.pFg->Release();
    gcap.pFg = NULL;
```

See Enumerate and Access Hardware Devices in DirectShow Applications for more information about device enumeration.

Building the Capture and Preview Filter Graph

AMCap includes a BuildCaptureGraph function that builds up a capture graph with both capture and preview components. Most applications will perform the same tasks sequentially as described in the following topics contained in this section.

- Set the Capture File Name
- Create a Graph Builder Object
- Set the Output File Name
- Retrieve the Current Filter Graph
- Add the Capture Filters to the Filter Graph
- Render the Capture Pins
- Render the Video Preview Pin
- Configure the Video Preview Window

These tasks are explained in greater detail later in this section.

AMCap also includes a BuildPreviewGraph function that is essentially a version of BuildCaptureGraph that deals only with preview. Another difference between

BuildCaptureGraph and BuildPreviewGraph is that the latter uses ICaptureGraphBuilder::SetFiltergraph to provide a filter graph object (IGraphBuilder interface) for the capture graph builder object (ICaptureGraphBuilder interface) to use. You probably won't need to call SetFiltergraph as the graph builder object creates a filter graph to use by default. Use this method only if you have already created your own filter graph and want the graph builder to use your filter graph. If you call this method after the graph builder has created a filter graph, this method will fail. BuildPreviewGraph calls CoCreateInstance to create a new filter graph object, if necessary, as shown in the following example.

```
hr = CoCreateInstance(CLSID_FilterGraph, NULL, CLSCTX_INPROC,
                      IID_IGraphBuilder, (LPVOID *)&gcap.pFg);


hr = gcap.pBuilder->SetFiltergraph(gcap.pFg);
    if (hr != NOERROR) {
      ErrMsg("Cannot give graph to builder");
      goto SetupPreviewFail;
    }
```

The details of each important task performed by BuildCaptureGraph follow.

Set the Capture File Name

AMCap's SetCaptureFile function displays the common Open File dialog box to enable the user to select a capture file. If the specified file is a new file, it calls the application-defined AllocCaptureFile function that prompts the user to allocate space for the capture file. This "preallocation" of file space is important, because it reserves a large block of space on disk. This speeds up the capture operation when it occurs, because the file space doesn't have to be allocated while capture takes place (it has been preallocated). The ICaptureGraphBuilder::AllocCapFile method performs the actual file allocation. IFileSinkFilter::SetFileName instructs the file writer filter to use the file name that the user chose. The code assumes you've called ICaptureGraphBuilder::SetOutputFileName to add the file writer to the filter graph. See Set the Output File Name for more information.

The AMCap-defined SetCaptureFile and AllocCaptureFile functions follow:

```
/*
 * Put up a dialog to allow the user to select a capture file.
 */
BOOL SetCaptureFile(HWND hWnd)
{
    if (OpenFileDialog(hWnd, gcap.szCaptureFile, _MAX_PATH)) {
        OFSTRUCT os;

      // We have a capture file name.

        /*
         * If this is a new file, then invite the user to
         * allocate some space.
         */
        if (OpenFile(gcap.szCaptureFile, &os, OF_EXIST) == HFILE_ERROR) {

          // Bring up dialog, and set new file size.
          BOOL f = AllocCaptureFile(hWnd);
          if (!f)
            return FALSE;
        }
    } else {
      return FALSE;
    }

    SetAppCaption();        // Need a new app caption.

    // Tell the file writer to use the new file name.
    if (gcap.pSink) {
        WCHAR wach[_MAX_PATH];
        MultiByteToWideChar(CP_ACP, MB_PRECOMPOSED, gcap.szCaptureFile, -1,
                            wach, _MAX_PATH);
        gcap.pSink->SetFileName(wach, NULL);
    }

    return TRUE;
}

// Preallocate the capture file.
//
BOOL AllocCaptureFile(HWND hWnd)
{
// We'll get into an infinite loop in the dlg proc setting a value.
    if (gcap.szCaptureFile[0] == 0)
      return FALSE;

    /*
     * Show the allocate file space dialog to encourage
     * the user to pre-allocate space.
     */
    if (DoDialog(hWnd, IDD_AllocCapFileSpace, AllocCapFileProc, 0)) {

      // Ensure repaint after dismissing dialog before
      // possibly lengthy operation.
      UpdateWindow(ghwndApp);

      // User has hit OK. Alloc requested capture file space.
      BOOL f = MakeBuilder();
      if (!f)
        return FALSE;
      WCHAR wach[_MAX_PATH];
      MultiByteToWideChar(CP_ACP, MB_PRECOMPOSED, gcap.szCaptureFile, -1,
                          wach, _MAX_PATH);
      if (gcap.pBuilder->AllocCapFile(wach,
                                gcap.wCapFileSize * 1024L * 1024L) != NOERROR) {
        MessageBoxA(ghwndApp, "Error",
                    "Failed to pre-allocate capture file space",
                    MB_OK | MB_ICONEXCLAMATION);
        return FALSE;
        }
    return TRUE;
    } else {
        return FALSE;
        }
}
```

Create a Graph Builder Object

AMCap's MakeBuilder function creates a capture graph builder object and obtains an ICaptureGraphBuilder interface pointer by calling CoCreateInstance. If you already have a capture graph builder object, you can call QueryInterface to obtain an interface pointer. AMCap stores the object pointer in the pBuilder member of the global gcap structure.

```
// Make a graph builder object we can use for capture graph building.
//
BOOL MakeBuilder()
{
    // We have one already.
    if (gcap.pBuilder)
      return TRUE;

    HRESULT hr = CoCreateInstance((REFCLSID)CLSID_CaptureGraphBuilder,
                    NULL, CLSCTX_INPROC, (REFIID)IID_ICaptureGraphBuilder,
                    (void **)&gcap.pBuilder);
    return (hr == NOERROR) ? TRUE : FALSE;
}
```

Set the Output File Name

AMCap creates the rendering section of the filter graph, consisting of the AVI MUX (multiplexer) and the File Writer. It also provides the filter graph with the previously specified file name to which to save the captured data. See About Capture Filter Graphs for more information about capture filter graph in general.

ICaptureGraphBuilder::SetOutputFileName signals to add the multiplexer and file writer to the filter graph, connects them, and sets the file name. The following example illustrates a call to SetOutputFileName.

```
//
// We need a rendering section that will write the capture file out in AVI
// file format.
//

    WCHAR wach[_MAX_PATH];
    MultiByteToWideChar(CP_ACP, MB_PRECOMPOSED, gcap.szCaptureFile, -1, wach,
                        _MAX_PATH);
    GUID guid = MEDIASUBTYPE_Avi;
    hr = gcap.pBuilder->SetOutputFileName(&guid, wach, &gcap.pRender,
                                          &gcap.pSink);
    if (hr != NOERROR) {
      ErrMsg("Error %x: Cannot set output file", hr);
      goto SetupCaptureFail;
    }
```

In the above code fragment the value of the first parameter, *pType*, in the call to SetOutputFileName is MEDIASUBTYPE_Avi, indicating that the capture graph builder object will insert an AVI multiplexer filter. Consequently, the file writer that is connected to the multiplexer will write the data to the capture file in AVI file format.

The second parameter, *lpwstrFile*, specifies the file name. The last two parameters contain pointers to the multiplexer filter and the file writer filter, respectively, and are initialized for you by the capture graph builder object upon return from SetOutputFileName. AMCap stores these pointers in the pRender and pSink members of its gcap structure. Internally, the capture graph builder object creates a filter graph object which exposes the IGraphBuilder interface and inserts these two filters into that filter graph. It tells the file writer to use the specified file when writing to disk.

Alternatively, if you want filters besides the multiplexer and file writer in the rendering section of your filter graph, call IFilterGraph::AddFilter to explicitly add the necessary filters. You might need to remember the pointer to the IBaseFilter interface of the first filter in your custom rendering chain so you can use it in calls such as RenderStream.

Retrieve the Current Filter Graph

Because the capture graph builder object created a filter graph in response to SetOutputFileName and you must put the necessary filters in the same filter graph, call the ICaptureGraphBuilder::GetFiltergraph method to retrieve the newly created filter graph. The pointer to the filter graph's IGraphBuilder interface is returned in the function's parameter.

```
//
// The graph builder created a filter graph to do that.  Find out what it is,
// and put the video capture filter in the graph too.
//

    hr = gcap.pBuilder->GetFiltergraph(&gcap.pFg);
    if (hr != NOERROR) {
      ErrMsg("Error %x: Cannot get filtergraph", hr);
      goto SetupCaptureFail;
    }
```

Add the Capture Filters to the Filter Graph

Call IFilterGraph::AddFilter to add the capture filters to the filter graph as shown in the following example.

```
    hr = gcap.pFg->AddFilter(gcap.pVCap, NULL);
    if (hr != NOERROR) {
      ErrMsg("Error %x: Cannot add vidcap to filtergraph", hr);
      goto SetupPreviewFail;
    }

    hr = gcap.pFg->AddFilter(gcap.pACap, NULL);
      if (hr != NOERROR) {
          ErrMsg("Error %x: Cannot add audcap to filtergraph", hr);
          goto SetupCaptureFail;
      }
```

Render the Capture Pins

The ICaptureGraphBuilder::RenderStream method connects the source filter's pin to the rendering filter. It connects intermediate filters if necessary. The pin category is optional, but typically specifies either a capture pin (PIN_CATEGORY_CAPTURE) or a preview pin (PIN_CATEGORY_PREVIEW). The following example connects the capture pin on the video capture filter (represented by the gcap.pVCap variable) to the renderer (represented by gcap.pRender).

```
//
// Render the video capture and preview pins - we may not have preview, so
// don't worry if it doesn't work.
//

    hr = gcap.pBuilder->RenderStream(&PIN_CATEGORY_CAPTURE, gcap.pVCap,
                                      NULL, gcap.pRender);
    // Error checking.
```

Call ICaptureGraphBuilder::RenderStream again to connect the audio capture filter (represented by gcap.pACap) to the audio renderer as in the following example.

```
//
// Render the audio capture pin?
//

    if (gcap.fCapAudio) {
      hr = gcap.pBuilder->RenderStream(&PIN_CATEGORY_CAPTURE, gcap.pACap, NULL, gca
    // Error checking.
```

Render the Video Preview Pin

Call ICaptureGraphBuilder::RenderStream again to render the graph from the capture filter's preview pin to a video renderer as in the following example.

```
    hr = gcap.pBuilder->RenderStream(&PIN_CATEGORY_PREVIEW, gcap.pVCap,
                                     NULL, NULL);
```

Configure the Video Preview Window

By default, the video preview window will be a separate window from your application window. If you want to change the default behavior, call ICaptureGraphBuilder::FindInterface to obtain a pointer to the IVideoWindow interface. The first parameter, *pCategory* specifies the output pin category to search for a connected filter that supports the desired interface. The code fragment below uses PIN_CATEGORY_PREVIEW to indicate a search beginning with all preview pins, and continuing to any pins and filters that connect to the preview pins. The second parameter, specified by the gcap.pVCap variable below, represents the video capture filter. The third (*riid*) is the identifier for the desired interface (IID_IVideoWindow), and the last will be filled upon return from this function to give you the **IVideoWindow** interface. After you have the **IVideoWindow** interface, you can call **IVideoWindow** methods such as put_Owner, put_WindowStyle, or SetWindowPosition to take ownership of the video preview window, make it a child of your application, or to position it as desired.

```
// This will go through a possible decoder, find the video renderer it's
// connected to, and get the IVideoWindow interface on it.
   hr = gcap.pBuilder->FindInterface(&PIN_CATEGORY_PREVIEW, gcap.pVCap,
                                     IID_IVideoWindow, (void **)&gcap.pVW);
    if (hr != NOERROR) {
      ErrMsg("This graph cannot preview");
    } else {
      RECT rc;
      gcap.pVW->put_Owner((long)ghwndApp);    // We own the window now.
      gcap.pVW->put_WindowStyle(WS_CHILD);    // You are now a child.
      // Give the preview window all our space but where the status bar is.
      GetClientRect(ghwndApp, &rc);
      cyBorder = GetSystemMetrics(SM_CYBORDER);
      cy = statusGetHeight() + cyBorder;
      rc.bottom -= cy;
      gcap.pVW->SetWindowPosition(0, 0, rc.right, rc.bottom); // Be this big.
      gcap.pVW->put_Visible(OATRUE);
    }
```

Now that you've built the entire capture filter graph, you can preview video, audio, or actually capture data.

Controlling the Capture Filter Graph

Because a capture filter graph constructed by the ICaptureGraphBuilder interface is simply a specialized filter graph, controlling a capture filter graph is much like controlling any other kind of filter graph: you use the IMediaControl interface's Run, Pause, and Stop methods. You can use the CBaseFilter::Pause method to cue things up, but remember that capture and recompression only happen when the graph is running. In addition, **ICaptureGraphBuilder** provides the ControlStream method to control the start and stop times of the capture filter graph's streams. Internally, **ControlStream** calls the IAMStreamControl::StartAt and IAMStreamControl::StopAt methods to start and stop the capture and preview portions of the filter graph for frame-accurate control.

Note: This method might not work on every capture filter because not every capture filter supports IAMStreamControl on its pins.

The ICaptureGraphBuilder::ControlStream method's first parameter (*pCategory*) is a pointer to a GUID that specifies the output pin category. This value is normally either PIN_CATEGORY_CAPTURE or PIN_CATEGORY_PREVIEW. See the Pin Property Set for a complete list of categories. Specify NULL to control all capture filters in the graph.

The second parameter (*pFilter*) in ICaptureGraphBuilder::ControlStream indicates which filter to control. Specify NULL to control the whole filter graph as AMCap does.

To run only the preview portion of the capture filter graph, prevent capture by calling ICaptureGraphBuilder::ControlStream with the capture pin category and the value MAX_TIME as the start time (third parameter, *pstart*). Call the method again with preview as the pin category, and a NULL start value to start preview immediately. The fourth parameter indicates the desired stop time (*pstop*, as with start time, NULL means immediately). MAX_TIME is defined in the DirectShow base classes as the maximum reference time, and in this case means to ignore or cancel the specified operation.

The last two parameters, *wStartCookie* and *wStopCookie* are start and stop cookies respectively. These cookies are arbitrary values set by the application so that it can differentiate between start and stop times and tell when specific actions have been completed. AMCap doesn't use a specific time in ICaptureGraphBuilder::ControlStream, so it doesn't need any cookies. If you use a cookie, use IMediaEvent to get event notifications. See IAMStreamControl for more information.

The following code fragment sets preview to start immediately, but ignores capture.

```
// Let the preview section run, but not the capture section.
// (There might not be a capture section.)
REFERENCE_TIME start = MAX_TIME, stop = MAX_TIME;
// Show us a preview first? but don't capture quite yet...
hr = gcap.pBuilder->ControlStream(&PIN_CATEGORY_PREVIEW, NULL,
                                  gcap.fWantPreview ? NULL : &start,
                                  gcap.fWantPreview ? &stop : NULL, 0, 0);
if (SUCCEEDED(hr))
    hr = gcap.pBuilder->ControlStream(&PIN_CATEGORY_CAPTURE, NULL, &start,
                                      NULL, 0, 0);
```

The same concept applies if you want only to capture and not preview. Set the capture start time to NULL to capture immediately and set the capture stop time to MAX_TIME. Set the preview start time to MAX_TIME, with an immediate (NULL) stop time.

The following example tells the filter graph to start the preview stream now (the *pstart* (third) parameter is NULL). Specifying MAX_TIME for the stop time (*pstop*) means disregard the stop time.

```
    gcap.pBuilder->ControlStream(&PIN_CATEGORY_PREVIEW, NULL, NULL, MAX_TIME, 0, 0)
```

Calling IMediaControl::Run runs the graph.

```
// Run the graph.
    IMediaControl *pMC = NULL;
    HRESULT hr = gcap.pFg->QueryInterface(IID_IMediaControl, (void **)&pMC);
    if (SUCCEEDED(hr)) {
      hr = pMC->Run();
      if (FAILED(hr)) {
          // Stop parts that ran.
          pMC->Stop();
      }
      pMC->Release();
    }
    if (FAILED(hr)) {
      ErrMsg("Error %x: Cannot run preview graph", hr);
      return FALSE;
```

If the graph is already running, start capture immediately with another call to ICaptureGraphBuilder::ControlStream. For example, the following call controls the whole filter graph (NULL *pFilter* (second) parameter), starts now (NULL *pstart* (third) parameter), and never stops (*pstop* (fourth) parameter initialized to MAX_TIME).

```
    REFERENCE_TIME stop = MAX_TIME;

    // NOW!
    gcap.pBuilder->ControlStream(&PIN_CATEGORY_CAPTURE, NULL, NULL, &stop, 0, 0);
```

AMCap uses this approach to start capture in response to the user clicking a button.

To stop the capture or preview operation, call IMediaControl::Stop, much as you called IMediaControl::Run to run the filter graph.

```
// Stop the graph.
    IMediaControl *pMC = NULL;
    HRESULT hr = gcap.pFg->QueryInterface(IID_IMediaControl, (void **)&pMC);
    if (SUCCEEDED(hr)) {
      hr = pMC->Stop();
      pMC->Release();
    }
```

Obtaining Capture Statistics

AMCap calls methods on the IAMDroppedFrames interface to obtain capture statistics. It determines the number of frames dropped (IAMDroppedFrames::GetNumDropped) and captured (IAMDroppedFrames::GetNumNotDropped), and uses the Win32 timeGetTime function at the beginning and end of capture to determine the capture operation's duration. The IAMDroppedFrames::GetAverageFrameSize method provides the average size of captured frames in bytes. Use the information from **IAMDroppedFrames::GetNumNotDropped**, **timeGetTime**, and **IAMDroppedFrames::GetAverageFrameSize** to obtain the total bytes captured and calculate the sustained frames per second for the capture operation.

Saving the Captured File

The original preallocated capture file temporarily holds capture data so you can capture as quickly as possible. When you want to save the data you captured to a more permanent location, call ICaptureGraphBuilder::CopyCaptureFile. This method transfers the captured data out of the previously allocated capture file to another file you choose. The resulting new file size matches the size of the actual captured data rather than the preallocated file size, which is usually very large.

The ICaptureGraphBuilder::CopyCaptureFile method's first parameter, *lpwstrOld*, is the file you're copying from (typically the very large, preallocated file you always use for capture). The second parameter, *lpwstrNew*, is the file to which you want to save your captured data. Setting the third parameter, *fAllowEscAbort*, to TRUE indicates that the user is allowed to abort the copy operation by pressing ESC. The last parameter, *pCallback*, is optional and enables you to supply a progress indicator, if desired, by implementing the IAMCopyCaptureFileProgress interface. The following example demonstrates a call to CopyCaptureFile.

```
    hr = pBuilder->CopyCaptureFile(wachSrcFile, wachDstFile,TRUE,NULL);
```

The SaveCaptureFile function defined by AMCap prompts the to enter a new file name in the Open File common dialog box, uses the Win32 MultiByteToWideChar function to convert the file name to a wide string, and saves the captured data to the specified file using ICaptureGraphBuilder::CopyCaptureFile.

```
/*
 * Put up a dialog to allow the user to save the contents of the capture file
 * elsewhere.
 */
BOOL SaveCaptureFile(HWND hWnd)
{
    HRESULT hr;
    char achDstFile[_MAX_PATH];
    WCHAR wachDstFile[_MAX_PATH];
    WCHAR wachSrcFile[_MAX_PATH];

    if (gcap.pBuilder == NULL)
      return FALSE;

    if (OpenFileDialog(hWnd, achDstFile, _MAX_PATH)) {

      // We have a capture file name.
      MultiByteToWideChar(CP_ACP, MB_PRECOMPOSED, gcap.szCaptureFile, -1,
                          wachSrcFile, _MAX_PATH);
      MultiByteToWideChar(CP_ACP, MB_PRECOMPOSED, achDstFile, -1,
                          wachDstFile, _MAX_PATH);
      statusUpdateStatus(ghwndStatus, "Saving capture file - please wait...");

      // We need our own graph builder because the main one might not exist.
```

```
        ICaptureGraphBuilder *pBuilder;
        hr = CoCreateInstance((REFCLSID)CLSID_CaptureGraphBuilder,
                              NULL, CLSCTX_INPROC, (REFIID)IID_ICaptureGraphBuilder,
                              (void **)&pBuilder);
        if (hr == NOERROR) {
            // Allow the user to press ESC to abort... don't ask for progress.
            hr = pBuilder->CopyCaptureFile(wachSrcFile, wachDstFile,TRUE,NULL);
            pBuilder->Release();
        }
        if (hr == S_OK)
            statusUpdateStatus(ghwndStatus, "Capture file saved");
        else if (hr == S_FALSE)
            statusUpdateStatus(ghwndStatus, "Capture file save aborted");
        else
            statusUpdateStatus(ghwndStatus, "Capture file save ERROR");
        return (hr == NOERROR ? TRUE : FALSE);

    } else {
return TRUE;    // They canceled or something.
    }
}
```

See Amcap.cpp and Status.cpp from the AMCap sample for more details about capturing media files and obtaining capture statistics.

Displaying Property Pages

DirectShow provides a number of interfaces to customize the settings of a capture filter graph including: **IAMStreamConfig**, **IAMVideoCompression**, **IAMCrossbar**, **IAMTVTuner**, **IAMTVAudio**, **IAMAnalogVideoDecoder**, **IAMCameraControl**, **IAMVideoProcAmp**. Creating a property page is one way of allowing users to interact with these settings.

To bring up the settings associated with an object on a property page, use an interface on the object to query for the ISpecifyPropertyPages interface. Use this interface to obtain a list of property page CLSIDs that this object supports. The CLSID list can be later passed to OleCreatePropertyFrame or OleCreatePropertyFrameIndirect to invoke a property sheet. This will supply your application with the custom property pages a filter has in addition to the standard pages.

There are at least 9 objects that can have property pages in capture applications. Capture applications usually have 2 of these objects at least; the video capture filter and the audio capture filter (call them pVCap and pACap). These objects expose the **IBaseFilter** interface which can be used to query for the ISpecifyPropertyPages interface. You can obtain a pointer to the other 7 objects as follows:

1. The video capture filter's capture pin. Get this by calling :

   ```
   FindInterface(&PIN_CATEGORY_CAPTURE, pVCap, IID_IPin, &pX);
   ```

2. The video capture filter's preview pin. Get this by calling:

   ```
   FindInterface(&PIN_CATEGORY_PREVIEW, pVCap, IID_IPin, &pX);
   ```

3. The audio capture filter's capture pin. Get this by calling:

   ```
   FindInterface(&PIN_CATEGORY_CAPTURE, pACap, IID_IPin, &pX);
   ```

4. The crossbar connected to the video capture filter. Get this by calling:

   ```
   FindInterface(NULL, pVCap, IID_IAMCrossbar, &pX);
   ```

5. The crossbar connected to the audio capture filter. This might be the same as object #4. Compare their IUnknown interfaces to find out. Get this by calling:

   ```
   FindInterface(NULL, pACap, IID_IAMCrossbar, &pX);
   ```

6. The TV Tuner connected to the video capture filter. Get this by calling:

   ```
   FindInterface(NULL, pVCap, IID_IAMTVTuner, &pX);
   ```

7. The TV Audio connected to the audio capture filter. Get this by calling:

   ```
   FindInterface(NULL, pACap, IID_IAMTVAudio, &pX);
   ```

If you do not wish to create your property page using the **ISpecifyPropertyPages** interface and the OleCreatePropertyFrame function, you can create your own custom property pages and use the results of your page to call the interfaces programmatically.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

About Cutlists

This article introduces cutlists and discusses interfaces that provide cutlist support. See Using Cutlists for information about the limitations of the current cutlist implementation and sample code for using cutlists in an application.

Contents of this article:

What Are Cutlists?

A *cutlist* is a list of audio or video clips (*cutlist elements*) you want to play back sequentially. For each clip, the cutlist element contains the file name from which to create the clip, and details about the clip including start and stop time within that file. A cutlist is either video- or audio- specific, and that video or audio data must all be of the same media type. The beginning time for the clip, relative to the source file, is called the *trimin* position and the ending time for the clip is the *trimout* position.

You use cutlists to edit pieces of AVI and WAV files together. For instance a video cutlist could contain video clips (elements) with characteristics as follows:

Clip #	File Name	Start Time	Stop Time	Type	Stream #
1	Venus.avi	5 seconds	10 seconds	video	0
2	Mars.avi	15 seconds	20 seconds	video	0
3	Venus.avi	15 seconds	30 seconds	video	0

In the preceding example, the first and third clips are both taken from the same file. One clip is from seconds 5 through 10 of Venus.avi, while another is from seconds 15 through 30 of the same file. Between those clips, the cutlist contains seconds 15 through 20 of Mars.avi. All clips are taken from the first video stream (stream 0) in their respective files. The clips play back sequentially (1, 2, and then 3).

Cutlist Objects and Interfaces

Microsoft® DirectShow™ defines the following objects that implement the specified interfaces. Applications use these objects and interfaces to create, manipulate, and play cutlists.

Object	Supported interfaces	Description
CLSID_SimpleCutList	IStandardCutList	Cutlist object
CLSID_VideoFileClip	IFileClip	Cutlist element (individual clip) object for video
CLSID_AudioFileClip	IFileClip	Cutlist element (individual clip) object for audio
CLSID_CutListGraphBuilder	ICutListGraphBuilder	Cutlist graph builder object

These interfaces enable application writers to construct filter graphs without having to worry about the specifics of each cutlist object. They provide a simple way to create and manipulate cutlists, and to create a filter graph to play an edited movie in real time. In addition, a single cutlist filter calls these interfaces — the application must be aware of the different cutlist filters that are installed and generate the proper filter graph.

If you need cutlist functionality that the preceding interfaces don't provide, such as detailed cutlist or cutlist element information, see the following interfaces.

- IAMAudioCutListElement
- IAMCutListElement
- IAMFileCutListElement
- IAMVideoCutListElement

Using Cutlists

This article summarizes the steps necessary to create and play a cutlist. It also lists the limitations of the current cutlist implementation and provides sample code for using cutlists in an application. See What Are Cutlists? for an introduction to cutlists.

Contents of this article:

To use cutlists, you must first explicitly include the cutlist.h header file in your project. Then you can create cutlist objects and use the interfaces they expose. The following list summarizes the steps for creating and playing back cutlists in your application. Creating a Video Cutlist presents the example code from this section as one unit rather than many separate fragments.

1. Create a standard cutlist object (CLSID_SimpleCutList) for each cutlist. Each cutlist can contain only one media type, so if you want both audio and video you must create one standard cutlist for all of your video clips, and one standard cutlist for all of your audio clips.

```
CoCreateInstance((REFCLSID)CLSID_SimpleCutList, NULL, CLSCTX_INPROC,
                 (REFIID)IID_IStandardCutList,(void**)&pVCutList);
```

2. Create a video file clip object (CLSID_VideoFileClip) or an audio file clip object (CLSID_AudioFileClip), as appropriate, for each stream in each AVI or WAV file you will use as the source for clips. If you want to play back both video and audio from the same file, you must still create separate video and audio file clip objects because the file clip objects are based upon one stream (either a video stream or an audio stream) within a file.

```
CoCreateInstance((REFCLSID)CLSID_VideoFileClip, NULL, CLSCTX_INPROC,
                 (REFIID)IID_IFileClip, (void**)&pVFileClip);
```

3. Tell the file clip object what file and stream in that file to use by calling the IFileClip::SetFileAndStream method as follows. The first video and audio streams (stream 0) are the only streams that DirectShow supports.

```
hr = pVFileClip->SetFileAndStream(L"jupiter.avi", 0);
```

4. Create one or more cutlist elements (individual clips) from each file clip by calling the IFileClip::CreateCut method. Each file clip represents a stream of data, and you can create more than one clip from each data stream. For instance, the example in What Are Cutlists? specifies two elements from Venus.avi, and one element from Mars.avi.
5. Add each cutlist element to the cutlist by calling the IStandardCutList::AddElement

method.

The following example creates two cutlist elements from the file clip and adds them to the cutlist. The file clip is the first video stream (stream zero) of Jupiter.avi, as specified previously in IFileClip::SetFileAndStream. The first element plays seconds 5 through 10 of Jupiter.avi and the second element plays second 0 through 5 of the same file.

```
hr = pVFileClip->CreateCut(&pVElement1,5*SCALE,10*SCALE,0,5*SCALE,0);
hr = pVCutList->AddElement(pVElement1,CL_DEFAULT_TIME,CL_DEFAULT_TIME);
hr = pVFileClip->CreateCut(&pVElement2,0,5*SCALE,0,5*SCALE,0);
hr = pVCutList->AddElement(pVElement2,CL_DEFAULT_TIME,CL_DEFAULT_TIME);
```

The first three parameters of IFileClip::CreateCut are the important ones. The first, *ppElement*, specifies the element, and is filled in for you. The second (*mtTrimIn*) and third (*mtTrimOut*) specify, respectively, the start and stop times for the clip relative to the original file (Jupiter.avi in this case). The last three parameters must always be zero, *mtTrimOut* minus *mtTrimIn*, and zero, respectively. A scale value of 10,000,000 scales the start and stop times to seconds.

The first parameter in the IStandardCutList::AddElement call, *pElement*, is the element obtained from the call to IFileClip::CreateCut. The last two parameters must be CL_DEFAULT_TIME to indicate that the element should be added to the end of the current cutlist, and its duration is the same as its duration in the original file.

6. Create a cutlist graph builder object (CLSID_CutListGraphBuilder) and use it to build a filter graph that will play your cutlists. Give it a standard cutlist object by using the ICutListGraphBuilder::AddCutList method, and then call the ICutListGraphBuilder::Render method to build a filter graph that can play the cutlist. The following code fragment illustrates these calls.

```
CoCreateInstance((REFCLSID)CLSID_CutListGraphBuilder,NULL,CLSCTX_INPROC,

                 (REFIID)IID_ICutListGraphBuilder,(void**)&pGraphBuilder);

.
.
.
// Give the cutlist to the graph builder
hr = pGraphBuilder->AddCutList(pVCutList, NULL);

// Tell the cutlist graph builder to build the graph
hr = pGraphBuilder->Render();
```

7. Play the cutlist filter graph and clean up resources as in the following example. Be sure to stop the graph before you remove the cutlist from the graph using ICutListGraphBuilder::RemoveCutList.

```
// Get the filter graph the builder just made
hr = pGraphBuilder->GetFilterGraph(&pGraph);

// Now get some useful graph interfaces
pGraph->QueryInterface(IID_IMediaControl,(void**)&pControl);
pGraph->QueryInterface(IID_IMediaEvent, (void**)&pEvent);
pGraph->Release();
```

```
// Run the graph
pControl->Run();

// Arbitrarily assumes 10000 millisecond timeout
pEvent->WaitForCompletion(10000, &lEventCode);
pControl->Stop();
pEvent->Release();
pControl->Release();

// Cleanup
hr = pGraphBuilder->RemoveCutList(pVCutList);
pVElement1->Release();
pVElement2->Release();
pVCutList->Release();
pVFileClip->Release();
pGraphBuilder->Release();
```

See the cutlist examples later in this article for more complete sample code illustrating these steps.

Cutlist Limitations

The following list discusses limitations that you should be aware of when using DirectShow's current cutlist implementation.

1. All clips in a cutlist must be the same format (media type).

   For video cutlists, this means that all the video clips must be of the same compression type, size, dimensions, bit depth, and so forth. In other words, all video clips in the cutlist must be represented by the same BITMAPINFOHEADER structure. For audio cutlists, this means they must all use the same compression format, sampling rate, bit depth, and number of channels. In other words all audio clips in the cutlist must be represented by the same WAVEFORMATEX structure.

   The first clip you add to a cutlist determines the cutlist's media type, and the media type required for all other clips you add to the cutlist. The IStandardCutList::AddElement method returns an invalid media type error (VFW_E_INVALIDMEDIATYPE) if you try to add a clip of a different media type to an existing cutlist.

2. All cuts must begin on a keyframe. If not, there will be an unwanted "fade in" effect at the cut point, instead of a clean switch from one clip to the next. The biggest limitation is that the first frame of the entire cutlist must be a keyframe. Otherwise, the file will be corrupt if you write the resulting cutlist to a file.
3. There is no way to save (persist) a cutlist. Every time your application runs, you must build the cutlist by hand. There is no file format for saving a cutlist you have previously constructed.
4. Audio cuts not made during silence might cause an audible "click" sound at the cut point if there is low to moderate volume and sparse audio at the cut point.
5. You can only create file clip objects from either the first video or audio stream of an AVI file. Extra streams in files with multiple video or audio streams are ignored.
6. WAV files and AVI files are the only types of files that you can use as source material for a cutlist. DirectShow doesn't support other formats, such as MPEG.
7. You can't identify WAV or AVI files used in cutlists by a universal network connection (UNC) network name. For example, the file name "x:\Venus.wav" is valid, but "\\Server\Share\Venus.wav" is not.
8. Cutlists with audio NULL elements (gaps in the audio track) can't be written correctly to a

file or played properly with the audio renderer included with DirectShow. Unless you have custom filters that can handle gaps in the audio stream, do not use audio NULL elements.

9. Cutlists work only with PCM audio, not compressed audio.
10. Cutlist support is currently not implemented for RLE compressed files.

Creating a Video Cutlist

The following code creates and plays a cutlist consisting of two video clips from one AVI file. It plays seconds 5 through 10 of the file followed by seconds 0 through 5. The code fragment contains no error checking for the sake of brevity. See Cutlist Sample Code (Simplecl) for an example that performs error checking.

```
HRESULT              hr;
ICutListGraphBuilder *pGraphBuilder;
IMediaControl        *pControl;
IMediaEvent          *pEvent;
IGraphBuilder        *pGraph;
IStandardCutList     *pVCutList;
IFileClip            *pVFileClip;
IAMCutListElement      *pVElement1, *pVElement2;
LONG                 lEventCode=0L;

CoInitialize(NULL);
// we need these 3 objects
CoCreateInstance((REFCLSID)CLSID_CutListGraphBuilder,NULL,CLSCTX_INPROC,
                     (REFIID)IID_ICutListGraphBuilder,(void**)&pGraphBuilder);
CoCreateInstance((REFCLSID)CLSID_VideoFileClip, NULL, CLSCTX_INPROC,
                     (REFIID)IID_IFileClip, (void**)&pVFileClip);
CoCreateInstance((REFCLSID)CLSID_SimpleCutList, NULL, CLSCTX_INPROC,
                     (REFIID)IID_IStandardCutList,(void**)&pVCutList);

// Tell the clip what file to use as a source file
hr = pVFileClip->SetFileAndStream(L"jupiter.avi", 0);

// Create some cutlist elements and add them to the standard cutlist
// from 5 to 10 seconds, then from 0 to 5 seconds
hr = pVFileClip->CreateCut(&pVElement1,5*SCALE,10*SCALE,0,5*SCALE,0);
hr = pVCutList->AddElement(pVElement1,CL_DEFAULT_TIME,CL_DEFAULT_TIME);
hr = pVFileClip->CreateCut(&pVElement2,0,5*SCALE,0,5*SCALE,0);
hr = pVCutList->AddElement(pVElement2,CL_DEFAULT_TIME,CL_DEFAULT_TIME);

// Give the cutlist to the graph builder
hr = pGraphBuilder->AddCutList(pVCutList, NULL);

// Tell the cutlist graph builder to build the graph
hr = pGraphBuilder->Render();

// Get the filter graph the builder just made
hr = pGraphBuilder->GetFilterGraph(&pGraph);

// Now get some useful graph interfaces
pGraph->QueryInterface(IID_IMediaControl,(void**)&pControl);
pGraph->QueryInterface(IID_IMediaEvent, (void**)&pEvent);
pGraph->Release();

// Run the graph
pControl->Run();

// Arbitrarily assumes 10000 millisecond timeout
pEvent->WaitForCompletion(10000, &lEventCode);
pControl->Stop();
```

```
pEvent->Release();
pControl->Release();

// Cleanup
hr = pGraphBuilder->RemoveCutList(pVCutList);
pVElement1->Release();
pVElement2->Release();
pVCutList->Release();
pVFileClip->Release();
pGraphBuilder->Release();

CoUninitialize();

// Exit
PostMessage(WM_QUIT, 0, 0);
```

The preceding example uses video only. The example in the next section uses both audio and video.

Creating Both Video and Audio Cutlists

The following code takes a file name from the command line and plays five different pieces of that AVI file back to back, with both sound and video synchronized. The code fragment contains no error checking for the sake of brevity. See Cutlist Sample Code (Simplecl) for an example that performs error checking.

```
HRESULT              hr;
ICutListGraphBuilder *pGraphBuilder;
IMediaControl        *pControl;
IMediaEvent          *pEvent;
IGraphBuilder        *pGraph;
IStandardCutList     *pVCutList, *pACutList;
IFileClip            *pAFileClip1;
IFileClip            *pVFileClip1;
IAMCutListElement      *pVElement1;
IAMCutListElement      *pVElement2;
IAMCutListElement      *pVElement3;
IAMCutListElement      *pVElement4;
IAMCutListElement      *pVElement5;
IAMCutListElement      *pAElement1;
IAMCutListElement      *pAElement2;
IAMCutListElement      *pAElement3;
IAMCutListElement      *pAElement4;
IAMCutListElement      *pAElement5;
LONG                 lEventCode=0L;
WCHAR                lpwstr[256];

CoInitialize(NULL);

CoCreateInstance((REFCLSID)CLSID_CutListGraphBuilder,NULL,CLSCTX_INPROC,
                 (REFIID)IID_ICutListGraphBuilder,(void**)&pGraphBuilder);

CoCreateInstance((REFCLSID)CLSID_AudioFileClip, NULL, CLSCTX_INPROC,
                 (REFIID)IID_IFileClip, (void**)&pAFileClip1);

CoCreateInstance((REFCLSID)CLSID_VideoFileClip, NULL, CLSCTX_INPROC,
                 (REFIID)IID_IFileClip, (void**)&pVFileClip1);

CoCreateInstance((REFCLSID)CLSID_SimpleCutList, NULL, CLSCTX_INPROC,
                 (REFIID)IID_IStandardCutList,(void**)&pVCutList);
CoCreateInstance((REFCLSID)CLSID_SimpleCutList, NULL, CLSCTX_INPROC,
                    (REFIID)IID_IStandardCutList,(void**)&pACutList);

// Get the Unicode file name to use from the command line
MultiByteToWideChar(CP_ACP, 0, m_lpCmdLine, strlen(m_lpCmdLine)+1,
                    lpwstr, sizeof(lpwstr)/sizeof(*lpwstr));

// tell the clips what file they are reading from
hr = pVFileClip1->SetFileAndStream(lpwstr, 0);
hr = pAFileClip1->SetFileAndStream(lpwstr, 0);

// Create some cuts and add them the cutlist

// from 2 to 6 seconds
hr = pVFileClip1->CreateCut(&pVElement1,2*SCALE,6*SCALE,0,4*SCALE,0);
hr = pVCutList->AddElement(pVElement1,CL_DEFAULT_TIME,CL_DEFAULT_TIME);
hr = pAFileClip1->CreateCut(&pAElement1,2*SCALE,6*SCALE,0,4*SCALE,0);
hr = pACutList->AddElement(pAElement1,CL_DEFAULT_TIME,CL_DEFAULT_TIME);

// from 20 to 24 seconds
hr = pVFileClip1->CreateCut(&pVElement2,20*SCALE,24*SCALE,0,4*SCALE,0);
hr = pVCutList->AddElement(pVElement2,CL_DEFAULT_TIME,CL_DEFAULT_TIME);
hr = pAFileClip1->CreateCut(&pAElement2,20*SCALE,24*SCALE,0,4*SCALE,0);
hr = pACutList->AddElement(pAElement2,CL_DEFAULT_TIME,CL_DEFAULT_TIME);

// from 65 to 69 seconds
hr = pVFileClip1->CreateCut(&pVElement3,65*SCALE,69*SCALE,0,4*SCALE,0);
hr = pVCutList->AddElement(pVElement3,CL_DEFAULT_TIME,CL_DEFAULT_TIME);
hr = pAFileClip1->CreateCut(&pAElement3,65*SCALE,69*SCALE,0,4*SCALE,0);
hr = pACutList->AddElement(pAElement3,CL_DEFAULT_TIME,CL_DEFAULT_TIME);

// from 35 to 39 seconds
hr = pVFileClip1->CreateCut(&pVElement4,35*SCALE,39*SCALE,0,4*SCALE,0);
hr = pVCutList->AddElement(pVElement4,CL_DEFAULT_TIME,CL_DEFAULT_TIME);
hr = pAFileClip1->CreateCut(&pAElement4,35*SCALE,39*SCALE,0,4*SCALE,0);
hr = pACutList->AddElement(pAElement4,CL_DEFAULT_TIME,CL_DEFAULT_TIME);

// from 12 to 16 seconds
hr = pVFileClip1->CreateCut(&pVElement5,12*SCALE,16*SCALE,0,4*SCALE,0);
hr = pVCutList->AddElement(pVElement5,CL_DEFAULT_TIME,CL_DEFAULT_TIME);
hr = pAFileClip1->CreateCut(&pAElement5,12*SCALE,16*SCALE,0,4*SCALE,0);
hr = pACutList->AddElement(pAElement5,CL_DEFAULT_TIME,CL_DEFAULT_TIME);

// Add the cutlists to the graph
hr = pGraphBuilder->AddCutList(pVCutList, NULL);
hr = pGraphBuilder->AddCutList(pACutList, NULL);

// Tell the cutlist graph builder to build the graph
hr = pGraphBuilder->Render();

// Get the filter graph the builder just made
hr = pGraphBuilder->GetFilterGraph(&pGraph);

// Now get some useful graph interfaces
pGraph->QueryInterface(IID_IMediaControl,(void**)&pControl);
pGraph->QueryInterface(IID_IMediaEvent, (void**)&pEvent);
pGraph->Release();

// Run the graph
pControl->Run();

// Arbitrarily assumes 10000 millisecond timeout
pEvent->WaitForCompletion(10000, &lEventCode);

pControl->Stop();
```

```
pEvent->Release();
pControl->Release();

// cleanup

// Remove the cutlist from the graph
hr = pGraphBuilder->RemoveCutList(pVCutList);
hr = pGraphBuilder->RemoveCutList(pACutList);

pVElement1->Release();
pVElement2->Release();
pVElement3->Release();
pVElement4->Release();
pVElement5->Release();
pAElement1->Release();
pAElement2->Release();
pAElement3->Release();
pAElement4->Release();
pAElement5->Release();

pACutList->Release();
pVCutList->Release();


pAFileClip1->Release();
pVFileClip1->Release();

pGraphBuilder->Release();

CoUninitialize();

// Exit
PostMessage(WM_QUIT, 0, 0);
```

The preceding example obtains video and audio clips from the same file. The next example adds a user interface and error checking, and it is available in the DirectShow SDK.

Cutlist Sample Code (Simplecl)

The Simplecl Sample (Cutlist Application) (Simplecl) from the DirectShow SDK demonstrates how to create and play back cutlists. By default, the DirectShow setup program installs Simplecl in the DXMedia\Samples\DS\cutlist\simplecl directory. Simplecl provides a File Open dialog box from which the user can chose a file to add to a cutlist. For each file, the user specifies a starting (trimin) position for the clip and an ending (trimout) position for the clip. For every AVI file specified, the sample tries to add the first video stream and the first audio stream to its respective cutlist. The user must add at least two files, and then can run the filter graph and see the clips played sequentially.

The DirectShow SDK also includes a sample that reads a list of cuts from a text file and plays them, much like Simplecl does. That sample, Cltext, is installed in the DXMedia\Samples\DS\cutlist\cltext directory by default.

The following code excerpts are from the Simplecl.h and Simplecl.cpp sample files. The sample includes error checking.

Simplecl.h declares a few global variables, including a ClipDetails structure to manage the user's file and clip start and stop time choices, and a ClipCollection structure to group the clip details. It also defines a SCALE constant to scale all user-specified times in one-second increments. The HELPER_RELEASE macro releases objects only if they exist, and then sets the object pointer to NULL to guard against releasing the same object multiple times. The following example contains fragments from Simplecl.h.

```
#define MAX_CLIPS 150
#define SCALE 10000000   // scale for 1 second of reference time

// Clip (element) details
struct ClipDetails
  {
    TCHAR szFilename[MAX_PATH];   // name of file containing clip
    REFERENCE_TIME start;         // Start (Trim In) position of clip within file
    REFERENCE_TIME stop;          // Stop (Trim Out) position of clip within file
  };

// Cutlist is a collection of clips (elements)
struct ClipCollection
  {
    int nNumClips;
    ClipDetails List[MAX_CLIPS];
  };

#define HELPER_RELEASE(x) { if (x) x->Release(); x = NULL; }

ClipCollection  gTheSet;            // Cutlist
```

The application initializes the user input structure as follows:

```
// ... in WinMain ...
ZeroMemory(&gTheSet, sizeof gTheSet);
```

Simplecl keeps track of the name of the media file that the user chooses as the source of a clip, tracks the number of files chosen, and displays a dialog box for the user to enter the start and stop times for each clip. The following code fragments relate to tracking the user input for clips:

```
// ... in WndMainProc ...

case IDM_ADDFILE:

  if (GetClipFileName(gTheSet.List[gTheSet.nNumClips].szFilename))

    { // Add file

      TCHAR szTitleBar[200];

      DialogBox(ghInst, MAKEINTRESOURCE(wDlgRes = IDD_MEDIATIMES),
                                ghApp, (DLGPROC)DialogProc);
      gTheSet.nNumClips = gTheSet.nNumClips + 1;
      wsprintf(szTitleBar, "SimpleCutList - %d clips(s) added.",
               gTheSet.nNumClips);
      SetWindowText(ghApp, szTitleBar);

    } // Add file

.
.
.
```

```
// ... in DialogProc ...
case IDOKTIMES:

  gTheList.List[gTheSet.nNumClips].start = GetDlgItemInt(h,
                                           IDC_TRIMIN2, NULL, FALSE);
  gTheList.List[gTheSet.nNumClips].stop = GetDlgItemInt(h,
                                          IDC_TRIMOUT2, NULL, FALSE);

  EndDialog(h,1);
  break;
```

The real work of the Simplecl sample is in the SimpleCutList function. If the user has chosen more than one clip, and then chooses Run from the Cutlist menu, then Simplecl builds and plays the cutlist. The following code checks the number of clips chosen, and calls SimpleCutList if more than one clip was chosen.

```
case IDM_RUN:
  if (gTheSet.nNumClips > 1)
    SimpleCutList();
  else
    DialogBox(ghInst, MAKEINTRESOURCE(wDlgRes = IDD_LESSTHAN2),
              ghApp, (DLGPROC)DialogProc);
  break;
```

After the user has entered the file and clip choices, the SimpleCutList function creates and plays the cutlist as follows:

```
  void SimpleCutList ()

    { // SimpleCutList //

      WCHAR wFile[MAX_PATH];  // File name

      // Initialize video and audio file clips and elements to NULL
      // so we can easily free objects later.
      for (int x = 0; x < MAX_CLIPS; ++x)

        {
          pVidFileClip[x] = NULL;
          pAudFileClip[x] = NULL;
            pVidCLElem[x] = NULL;
            pAudCLElem[x] = NULL;
        };

      // Create cutlist graph builder object
      hr = CoCreateInstance(CLSID_CutListGraphBuilder, NULL,
                            CLSCTX_INPROC, IID_ICutListGraphBuilder,
                            (void**)&pCLGraphBuilder);

      if (FAILED(hr))
        { // CoCreateInstance of CutListGraphBuiler failed
          MessageBox(ghApp,
                     "CoCreateInstance of CutListGraphBuiler failed",
                     APPLICATIONNAME, MB_OK);
          TearDownTheGraph();
          return;
        } // CoCreateInstance of CutListGraphBuiler failed

      // Create simple (standard) cutlist object for video
      hr = CoCreateInstance(CLSID_SimpleCutList, NULL,
                          CLSCTX_INPROC, IID_IStandardCutList,
                          (void**)&pVideoCL);

    if (FAILED(hr))
      { // CoCreateInstance of video SimpleCutList failed
        MessageBox(ghApp,
                   "CoCreateInstance of video SimpleCutList failed",
                   APPLICATIONNAME, MB_OK);
        TearDownTheGraph();
        return;
      } // CoCreateInstance of video SimpleCutList failed

    // Create simple (standard) cutlist object for audio
    hr = CoCreateInstance(CLSID_SimpleCutList, NULL,
                          CLSCTX_INPROC, IID_IStandardCutList,
                          (void**)&pAudioCL);

    if (FAILED(hr))
      { // CoCreateInstance of audio SimpleCutList failed
        MessageBox(ghApp,
                   "CoCreateInstance of audio SimpleCutList failed",
                   APPLICATIONNAME, MB_OK);
        TearDownTheGraph();
        return;
      } // CoCreateInstance of audio SimpleCutList failed

    // Create the individual clips and add them to the cutlist
    nVidElems = nAudElems = 0;
    for (x = 0; x < gTheSet.nNumClips; ++x)

      { // Individual clips

        MultiByteToWideChar(CP_ACP, 0,
                            gTheSet.List[x].szFilename,
                            -1, wFile, MAX_PATH );

        // Create a video clip object and give it the file and stream
        // to read from.
        // SetFileAndStream will fail if we call it from a video clip
        // object and the clip is not a video clip.
        hr = CoCreateInstance(CLSID_VideoFileClip, NULL,
                              CLSCTX_INPROC, IID_IFileClip,
                              (void**)&pVidFileClip[nVidElems]);

        hr = pVidFileClip[nVidElems]->SetFileAndStream(wFile, 0);

        if (SUCCEEDED(hr))

          { // Create video cut and add the clip (element) to the cutlist

            hr = pVidFileClip[nVidElems]->CreateCut(&pVidCLElem[nVidElems],
                    gTheSet.List[x].start*SCALE,
                    gTheSet.List[x].stop*SCALE,
                    0,
                    (gTheSet.List[x].stop-gTheSet.List[x].start)*SCALE,
                    0);

            if (SUCCEEDED(hr))

              { // Add the element to the cutlist

                hr = pVideoCL->AddElement(pVidCLElem[nVidElems], CL_DEFAULT_TIME,
          if (SUCCEEDED(hr))
            ++nVidElems;

          else

            { // AddElement failed so release associated objects

              HELPER_RELEASE(pVidCLElem[nVidElems]);
              HELPER_RELEASE(pVidFileClip[nVidElems]);
              MessageBox(ghApp, "AddElement (video) failed!", APPLICATIONNA

            } // AddElement failed so release associated objects
        } // Add the element to the cutlist

      else MessageBox(ghApp, "CreateCut (video) failed!", APPLICATIONNAME,

    } // Create video cut

  else

    { // Problems creating video stream

      HELPER_RELEASE(pVidFileClip[nVidElems]);
      MessageBox(ghApp, "SetFileAndStream (video) failed!", APPLICATIONNAME

    } // Problems creating video stream

  // Create an audio clip object and give it the file and stream
  // to read from.
  // SetFileAndStream will fail if we call it from an audio clip
  // object and the clip is not an audio clip
  hr = CoCreateInstance(CLSID_AudioFileClip, NULL,
                        CLSCTX_INPROC, IID_IFileClip,
                        (void**)&pAudFileClip[nAudElems]);

  hr = pAudFileClip[nAudElems]->SetFileAndStream(wFile, 0);

  if (SUCCEEDED(hr))

    { // Create audio cut and add the clip (element) to the cutlist

      hr = pAudFileClip[nAudElems]->CreateCut(&pAudCLElem[nAudElems],
              gTheSet.List[x].start*SCALE,
              gTheSet.List[x].stop*SCALE,
              0,
              (gTheSet.List[x].stop-gTheSet.List[x].start)*SCALE,
              0);

      if (SUCCEEDED(hr))

        { // Add the element to the cutlist

          hr = pAudioCL->AddElement(pAudCLElem[nAudElems],
                                    CL_DEFAULT_TIME,
                                    CL_DEFAULT_TIME);

          if (SUCCEEDED(hr))
            ++nAudElems;

          else

           { // AddElement failed so release associated objects

              HELPER_RELEASE(pAudCLElem[nAudElems]);
```

```
                  HELPER_RELEASE(pAudFileClip[nAudElems]);
                  MessageBox(ghApp, "AddElement (audio) failed!", APPLICATIONNA

                } // AddElement failed so release associated objects

              } // Add the element to the cutlist

            else MessageBox(ghApp, "CreateCut (audio) failed!", APPLICATIONNAME,

          } // Create audio cut

        // Problems creating audio stream
        else

          { // Problems creating audio stream

            HELPER_RELEASE(pAudFileClip[nAudElems]);
            MessageBox(ghApp, "SetFileAndStream (audio) failed!", APPLICATIONNAME

          } // Problems creating audio stream

      } // Individual clips

    // Add the video cutlist to the filter graph
    hr = pCLGraphBuilder->AddCutList(pVideoCL, NULL);

    if (FAILED(hr)) // AddCutList (video) failed
        MessageBox(ghApp, "AddCutList (video) failed", APPLICATIONNAME, MB_OK);

    // Add the audio cutlist to the filter graph
    hr = pCLGraphBuilder->AddCutList(pAudioCL, NULL);

    if (FAILED(hr)) // AddCutList (audio) failed
        MessageBox(ghApp, "AddCutList (audio) failed", APPLICATIONNAME, MB_OK);

    if ((!pVideoCL) && (!pAudioCL))

      { // Clean up

        TearDownTheGraph();
        return;

      } // Clean up

    // Let the filter graph manager construct the appropriate graph
    // automatically
    hr = pCLGraphBuilder->Render();

    if (FAILED(hr))
      { // Problems rendering the graph
        if (!AMGetErrorText(hr, gszScratch, 2048))
          MessageBox(ghApp, "Problems rendering the graph!", APPLICATIONNAME, MB_
        else
          MessageBox(ghApp, gszScratch, APPLICATIONNAME, MB_OK);
        TearDownTheGraph();
        return;
      } // Problems rendering the graph

    // Retrieve the filter graph and useful interfaces
    hr = pCLGraphBuilder->GetFilterGraph(&pigb);

    if (FAILED(hr))
      { // Problems retrieving the graph pointer
        if (!AMGetErrorText(hr, gszScratch, 2048))
```

```
          MessageBox(ghApp, "Problems retrieving the graph pointer!", APPLICATION
        else
          MessageBox(ghApp, gszScratch, APPLICATIONNAME, MB_OK);
        TearDownTheGraph();
        return;
      } // Problems retrieving the graph pointer

    // QueryInterface for some basic interfaces
    pigb->QueryInterface(IID_IMediaControl, (void **)&pimc);
    pigb->QueryInterface(IID_IMediaEventEx, (void **)&pimex);
    pigb->QueryInterface(IID_IVideoWindow, (void **)&pivw);

    // Decrement the ref count on the filter graph
    pigb->Release();

    // Prepare to play in the main application window's client area

    RECT rc;
    GetClientRect(ghApp, &rc);
    hr = pivw->put_Owner((OAHWND)ghApp);
    hr = pivw->put_WindowStyle(WS_CHILD|WS_CLIPSIBLINGS);
    hr = pivw->SetWindowPosition(rc.left, rc.top, rc.right, rc.bottom);

    // Have the graph signal event via window callbacks for performance
    pimex->SetNotifyWindow((OAHWND)ghApp, WM_GRAPHNOTIFY, 0);

    // Run the graph if RenderFile succeeded
    pimc->Run();

  } // SimpleCutList //
```

Simplecl's TearDownTheGraph function releases all objects and cleans up as follows.

```
void TearDownTheGraph (void)

  { // TearDownTheGraph //

    if (pimc)
      pimc->Stop();

    if (pivw)

      { // Hide the playback window first thing

        pivw->put_Visible(OAFALSE);
        pivw->put_Owner(NULL);

      } //

    HELPER_RELEASE(pimex);
    HELPER_RELEASE(pimc);
    HELPER_RELEASE(pivw);

    // Remove the video cutlist from the filter graph to free resources
    if (pCLGraphBuilder && pVideoCL)
      pCLGraphBuilder->RemoveCutList(pVideoCL);

    // Remove the audio cutlist from the filter graph to free resources
    if (pCLGraphBuilder && pAudioCL)
      pCLGraphBuilder->RemoveCutList(pAudioCL);

    for (int x = 0; x < nAudElems; ++x)
```

```
        { // Release audio objects

          HELPER_RELEASE(pAudCLElem[x]);
          HELPER_RELEASE(pAudFileClip[x]);

        } // Release audio objects

      for (x = 0; x < nVidElems; ++x)

        { // Release video objects

          HELPER_RELEASE(pVidCLElem[x]);
          HELPER_RELEASE(pVidFileClip[x]);

        } // Release video objects

      HELPER_RELEASE(pVideoCL);
      HELPER_RELEASE(pAudioCL);
      HELPER_RELEASE(pCLGraphBuilder);

    } // TearDownTheGraph //
```

DVD for Title Vendors

DVD-Video discs typically contain programs such as feature films, interactive games, or video reference materials like encyclopedias. The end user can play back those programs on a DVD-Video player or on a DVD-ROM-equipped computer. Some of the features of DVD-Video include support for multiple languages, parental control, different camera angles, and closed captioning.

This article discusses the unique features of DVD that are not available in pure MPEG-2 (its parent format) and outlines the interfaces and methods DirectShow provides in support of those features.

DVD-unique features include the following:

- Better seeking
- Subpictures
- Multiple language support
- Variable speed play (forward/backward scan)
- Consumer DVD interactivity
- Seamless video angle change
- Parental control

Title vendors can create feature-rich applications by taking advantage of these DVD-Video features.

Note This release of DirectShow supports DVD-Video. It does not support pure MPEG-2.

See Additional DVD Resources on the Web for a list of DVD resources on the Web.

Contents of this article:

DVD Interfaces

DirectShow provides the following DVD-related interfaces.

Interface	**Purpose**
IDvdGraphBuilder	Allows the DVD application writer to easily build a filter graph for DVD-Video playback.
IDvdControl	Controls the playback and search mechanisms of a DVD-Video disc that contains one or more video movies.
IDvdInfo	Allows an application to query for attributes of available DVD-Video titles and the DVD player status. It also allows for control of a DVD player beyond Annex J in the DVD specification.

Later sections of this article group methods from these interfaces into functional categories.

DirectShow also provides a number of events. See DVD Events for more information.

DVD Control Data Structure

DVD-Video contains a nested hierarchy that provides search capabilities at several levels in the DVD data. This nested "control data" points to the real video and audio data. The following table outlines the structure of the control data for a DVD-Video volume. Each DVD volume can contain from one to 99 video title sets, which can contain one or more titles, which can contain one or more program chains. This nested structure continues to the smallest unit, which is the "pack." DirectShow provides seeking capabilities for DVD at three distinct levels, as outlined in Seeking in DVD.

DVD-Video Volume Structure

Control Data	**Description**
Video Title Set (VTS)	Collection of movies. A single volume can contain one to 99 video title sets.
Title	Individual movie. This may be a simple linear movie, consisting of one program chain, or it might consist of several program chains.
Program Chain (PGC)	A collection of programs (often chapters in a movie).
Chapter/Part of Title (PTT)	Collection of programs. Can delimit scenes or provide optional scenes from which to choose. Possible options include different ratings, camera angles, or a different storyline.
Program (PG)	Collection of cells, which normally make up a scene.
Cell	Collection of Video Object Units. Typically all the video and audio data from a certain number of Video Object Units.
Cell-Part	Stream of data (multi-angle only).
Video Object Unit (VOBU)	Usually half a second of video.
Pack	2KB of data, consisting of only one media type (such as video or audio).

Pure MPEG-2 supports only the title and pack from the list above.

DVD Features

This section outlines features specific to DVD-Video and lists the DirectShow methods that provide these features.

Seeking in DVD

DirectShow enables you to seek at several different levels within the DVD content. Because pure MPEG-2 supports only title and pack control data, it does not provide the flexibility in seeking that DVD does.

The following table shows the DirectShow DVD methods for seeking at various levels.

Seeking level	**Control data**	**IDvdControl methods**
Title Seeks	Video Title Set (VTS), Title, Program Chain (PGC)	TitlePlay
Chapter Seeks	Chapter/Part of Title (PTT), Program (PG), Cell	ChapterPlay (specifying title and chapter number), ChapterSearch (search for a chapter within the same title), PrevPGSearch, TopPGSearch, NextPGSearch
Time Seeks	Cell-Part, Video Object Unit (VOBU), Pack	TimePlay (start playing specified title from specified time), TimeSearch (start playing from specified time within the same title)

Subpicture

Subpicture is an extra media type that is decoded and alpha blended. The data on the alpha channel could be text for closed-captioning, buttons to provide a user interface, menus, subtitles, credits, and so on.

Methods relating to subpicture include the following:

- IDvdControl::SubpictureStreamChange
- IDvdInfo::GetCurrentSubpicture
- IDvdInfo::GetSubpictureLanguage
- IDvdInfo::GetCurrentSubpictureAttributes

Multiple Language Support

DVD-Video provides support of up to eight audio tracks to accommodate various languages. It also supports text in different languages for statistics related to the DVD title such as cast, crew, or title.

Methods relating to language support include the following:

- IDvdControl::MenuLanguageSelect
- IDvdInfo::GetAudioLanguage
- IDvdInfo::GetSubpictureLanguage

Variable Speed Play

DirectShow provides variable speed play through the IDvdControl::ForwardScan and IDvdControl::BackwardScan methods:

Consumer DVD Interactivity

The consumer of a DVD title can interact with the title by selecting and activating buttons, displaying menus, and using the mouse to select and activate buttons.

Methods relating to consumer interactivity include the following:

- IDvdControl::MenuCall
- IDvdControl::UpperButtonSelect
- IDvdControl::LowerButtonSelect
- IDvdControl::LeftButtonSelect
- IDvdControl::RightButtonSelect
- IDvdControl::ButtonActivate
- IDvdControl::ButtonSelectAndActivate
- IDvdControl::MouseActivate
- IDvdControl::MouseSelect
- IDvdInfo::GetCurrentButton

Seamless Video Angle Change

DVD-Video supports up to nine camera angles. These angles can be completely independent video streams, or different camera angles of the same scene. The fast seeking of the DVD disc allows switching angles seamlessly.

Methods relating to video angles include IDvdControl::AngleChange and IDvdInfo::GetCurrentAngle.

Parental Control

Parental control provides security for parents who want to prevent children from viewing certain types of content. Content might be authored at a particular level, or might contain the same scene shot at different rating levels to provide a viewing alternative for children.

Methods relating to parental control include the following:

- IDvdControl::ParentalLevelSelect
- IDvdControl::ParentalCountrySelect
- IDvdInfo::GetPlayerParentalLevel
- IDvdInfo::GetTitleParentalLevels

Additional DVD Resources on the Web

The following list contains links to a few of the Web sites that provide DVD information. Search the Web for other DVD resources. Note that most of these external links point to servers that are not under Microsoft's control. Please read Microsoft's official statement regarding other servers.

- http: //www.microsoft.com/hwdev/devdes/dvdwp.htm contains a whitepaper titled "DVD and Microsoft Operating Systems" which outlines the planned support for DVD under future Windows operating systems.
- ☐ http://www.unik.no/~robert/hifi/dvd/ includes links to many other DVD sites, news stories, and other resources.
- ☐ http://reality.sgi.com/nemec/dvd.html contains notes from a DVD technical forum.
- ☐ http://www.c-cube.com/technology/dvd.html contains a whitepaper on DVD.
- ☐ http://www.icdia.org/dvdfaq02.html contains a DVD frequently asked questions list.
- ☐ http://www.videodiscovery.com/vdyweb/dvd/dvdfaq.html contains a DVD frequently asked questions list which is also available from alt.video.dvd Usenet newsgroup.

Previous Home Topic Contents Index Next

Filter Developer's Guide

If you are developing a filter for use in a DirectShow filter graph, read the articles in this section.

How to...

This section gives step-by-step procedures for writing and using different kinds of filters, including a video capture filter, an audio capture filter, and a transform filter.

Write a Video Capture Filter

This article outlines important points to consider when writing a video capture filter. The Microsoft® DirectShow™ SDK includes a standard VFW Video Capture filter.

Contents of this article:

Capture and Preview Pin Requirements

The capture pin and preview pin (if there is one) of the capture filter must support the IKsPropertySet interface. Applications call this interface to ask "what category of pin are you?" by getting the AMPROPERTY_PIN_CATEGORY value of the AMPROPSETID_Pin property set. The value you return is typically either the PIN_CATEGORY_CAPTURE or PIN_CATEGORY_PREVIEW GUID. (See Pin Property Set for a complete list of pin categories.) A capture filter must support **IKsPropertySet** or an application can't tell how to connect the filter in a filter graph.

You can name the pin anything you want and you can have other output pins for any additional purposes that you want. If your pin name begins with the tilde (~) character, the filter graph will not automatically render that pin when an application calls IGraphBuilder::RenderFile. For instance, if you have a capture filter with both a capture pin and a preview pin, you might want to name the capture pin "~capture" and the preview pin "preview." Given those names, if an application renders that filter in a graph, the preview pin will be connected to a video renderer, and nothing will be connected to the capture filter, which is probably what you want to happen by default. This can also apply to pins that are just informational and are not meant to be rendered, or need to be enumerated so that their properties can be set.

The tilde (~) prefix only affects the behavior of RenderFile and intelligent connect (IGraphBuilder::Connect). Note that intelligent connect can still be used to connect pins with this property if they implement the IPin::Connect method. However, output pins of intermediate filters which are being used to complete the connection which have the tilde at the start of their name will not be connected as part of the intelligent connection attempt.

See Audio Capture Pin Requirements for more details about audio capture filters.

The following sample code demonstrates how to implement IKsPropertySet on a capture pin.

```
//
```

```
// PIN CATEGORIES - let the world know that we are a CAPTURE pin
//

HRESULT CMyCapturePin::Set(REFGUID guidPropSet, DWORD dwPropID, LPVOID pInstanceDat
{
    return E_NOTIMPL;
}

// To get a property, the caller allocates a buffer which the called
// function fills in. To determine necessary buffer size, call Get with
// pPropData=NULL and cbPropData=0.
HRESULT CMyCapturePin::Get(REFGUID guidPropSet, DWORD dwPropID, LPVOID pInstanceDat
{
    if (guidPropSet != AMPROPSETID_Pin)
        return E_PROP_SET_UNSUPPORTED;

    if (dwPropID != AMPROPERTY_PIN_CATEGORY)
        return E_PROP_ID_UNSUPPORTED;

    if (pPropData == NULL && pcbReturned == NULL)
        return E_POINTER;

    if (pcbReturned)
        *pcbReturned = sizeof(GUID);

    if (pPropData == NULL)
        return S_OK;

    if (cbPropData < sizeof(GUID))
        return E_UNEXPECTED;

    *(GUID *)pPropData = PIN_CATEGORY_CAPTURE;
    return S_OK;
}


// QuerySupported must either return E_NOTIMPL or correctly indicate
// if getting or setting the property set and property is supported.
// S_OK indicates the property set and property ID combination is
HRESULT CMyCapturePin::QuerySupported(REFGUID guidPropSet, DWORD dwPropID, DWORD *p
{
    if (guidPropSet != AMPROPSETID_Pin)
        return E_PROP_SET_UNSUPPORTED;

    if (dwPropID != AMPROPERTY_PIN_CATEGORY)
        return E_PROP_ID_UNSUPPORTED;

    if (pTypeSupport)
        *pTypeSupport = KSPROPERTY_SUPPORT_GET;
    return S_OK;
}
```

Optimizing Capture Versus Preview (Optional)

When your filter is running and capturing data, you must send a copy of the frame from your preview pin as well as from your capture pin. If you can do hardware-assisted preview — through an overlay, for example — and if you have a preview pin, you can use the IOverlay interface transport for your preview pin instead of the IMemInputPin interface. Using **IOverlay** is optional. If you can't do hardware-assisted preview, only send a frame out the preview pin if you have some spare time. Don't do it if it will make you drop any frames — the capture pin has priority.

For example, you might deliver a frame from the preview pin only if you have nothing to send from the capture pin right now and the downstream filter has released all buffers previously delivered from the capture pin.

If you can capture only one format of data, and the preview and capture pins must therefore produce the same media type, or if you want information about how to properly reconnect pins, read on. Otherwise, skip this section.

Send data of the same format from the preview and capture pins. If the filter graph manager reconnects your capture pin with a different format, you must reconnect your preview pin with the same format to make it work. If your capture pin is connected but your preview pin is not, you must allow only your preview pin to connect with the same media type as the capture pin. They must match.

Note: If your preview pin is producing 8-bit RGB and must reconnect using 16-bit RGB, the reconnect might fail. This failure might occur if you are connected to a video renderer, because the renderer might need a color converter filter inserted between the filters to convert the 16-bit RGB to 8-bit RGB. In this case, calling the IFilterGraph::Reconnect method will fail. You must do a full-fledged connect again (with CBasePin::Connect). If you only change between different sizes of motion JPEG, don't worry; a simple reconnect will always work.

The following sample code shows how the more complicated reconnection would work.

```
// Capture pin is being told to use a certain media type
//
CCapturePin::SetMediaType(CMediaType *pmt);
{
    if (m_pMyPreviewPin->IsConnected()) {

        // We need to reconnect our preview pin with this media type
        if (m_pMyPreviewPin->GetConnected()->QueryAccept(pmt) == NOERROR) {

                // The other filter that the preview pin is connected to
                // can accept this new media type, so we simple reconnect
                m_pGraph->Reconnect(m_pMyPreviewPin);
        } else {
                // The other filter WON'T accept this new time. Time to do
                // the connection all over again, possibly pulling in new
                // filters to help connect them
                IPin *pPin = m_pMyPreviewPin->GetConnected();
                m_pGraph->Disconnect(pPin);     // disconnect upstream first
                m_pGraph->Disconnect(m_pMyPreviewPin);
                // The sample code below will make sure the new connection
                // happens with the same media type as we are using
                hr = m_pGraph->Connect(m_pMyPreviewPin, pPin);
                if (FAILED(hr))
                        ; // UH OH !!!
        }
    }
}

CPreviewPin::CheckMediaType(CMediaType *pmt)
{
        CMediaType cmt = m_pMyCapturePin->m_mt;
        if (m_pMyCapturePin->IsConnected() && *pmt != cmt)
                // Sorry, our preview pin is only allowed to connect with
                // the same format as the capture pin
                return E_INVALIDARG;

        else if (!m_pMyCapturePin->IsConnected())
                // You decide if you like this media type or not, maybe by
                // knowing what the capture pin will connect with. But don't
                // worry, when the capture pin is connected, we will be
                // reconnected to use the same format

        // if our capture pin is connected, and this is the same media type,
        // we are OK.
        return NOERROR;
}
```

Registering a Video Capture Filter

You must register your filter in the video capture filter category. See AMovieDllRegisterServer2 for more information.

Producing Data

Produce data on capture and preview pins only when the filter graph is in a running state. You do not send data from your pins when the filter graph is paused. This will confuse the filter graph unless you return VFW_S_CANT_CUE from the CBaseFilter::GetState function, warning the filter graph that you do not send data when paused. The following code shows you what to do.

```
CMyVidcapFilter::GetState(DWORD dw, FILTER_STATE *State)
{
        *State = m_State;
        if (m_State == State_Paused)
                return VFW_S_CANT_CUE;
        else
                return S_OK;
}
```

Controlling Individual Streams

All output pins should support the IAMStreamControl interface, so an application can turn each pin on or off individually (for instance, to preview without capturing). **IAMStreamControl** enables you to switch between preview and capture without rebuilding a different graph.

Time Stamping

When you capture a frame and are sending it, time stamp the frame with the time the graph's clock says it is when the frame is captured. The end time is the start time plus the duration. For example, if you are capturing at 10 frames per second, and the graph's clock says 200,000,000 units at the time the frame is captured, the sample is stamped (200000000, 201000000) (there are 10,000,000 units per second).

A preview frame should have no time stamp because of latency problems. The latency is due to the fact that, if the time of the sample is the graph's time when it leaves the preview pin, by the time the sample gets to the renderer, it will be late. Therefore the renderer may choose not to draw the sample in order to save time and "catch up", which can't happen for a live stream. Implementing IAMStreamControl requires time stamps, so you can choose not to implement stream control on the preview pin, only time stamp the preview pin sample when there are outstanding requests to start or stop, or live with the latency problem. See the source code for the VidCap Sample (Video Capture Filter) sample for details.

You should set the media time of the sample you deliver; also set the regular time stamp for your capture pin. The media time is the frame number of the sample. For example, if you are capturing and sending frames and frame 3 gets dropped, you would set the media time values to be (0,1) (1,2) (2,3) (4,5) (5,6) and so on. This informs the downstream filters if any video frames were dropped even when the regular time stamps are a little random because the clock being used is not the video digitizing clock.

Also, if you are in a running state, and then pause, and then run again, you must not send a sample with a time stamp less than the last one you sent before pausing. Time stamps can never go back in time, not even back to before a pause occurred.

Necessary Interfaces

Read about the following interfaces and consider implementing them. You should implement these interfaces to provide functionality that applications might rely on, so these interfaces are strongly recommended.

- Implement IAMDroppedFrames on your filter or on each output pin that sends data.
- Implement IAMStreamConfig on each output pin that sends video data.
- Implement IAMStreamControl on each output pin that sends data.
- Implement IAMVideoCompression on each output pin that sends video data.

Write an Audio Capture Filter

This article outlines important points to consider when writing an audio capture filter. The Microsoft® DirectShow™ SDK includes a standard Audio Capture filter.

Contents of this article:

Audio Capture Pin Requirements

The capture filter's capture pin and preview pin (if there is one) must support the

IKsPropertySet interface. See Capture and Preview Pin Requirements for more details and sample code for implementing **IKsPropertySet** on your capture pin.

You must have one input pin for every sound source the capture card can mix before it digitizes the audio. For instance, if your sound card has a line in, microphone in, and CD-ROM input, you would have three input pins. You don't typically connect these input pins to any other filters — you just support the IAMAudioInputMixer interface on each pin and an application will set recording levels, balance, treble, and so on, on each pin using that interface.

Registering an Audio Capture Filter

You must register your filter in the audio capture filter category. See the AMovieDllRegisterServer2 function for more information.

Producing Data

Produce data on the capture pin only when the filter graph is in a running state. Do not send data from your pins when the filter graph is paused. This will confuse the filter graph unless you return VFW_S_CANT_CUE from the CBaseFilter::GetState function, which warns the filter graph that you do not send data when paused. The following code sample shows how to do this.

```
CMyVidcapFilter::GetState(DWORD dw, FILTER_STATE *State)
{
        *State = m_State;
        if (m_State == State_Paused)
                return VFW_S_CANT_CUE;
        else
                return S_OK;
}
```

Controlling Individual Streams

All output pins should support the IAMStreamControl interface, so an application can turn each pin on or off individually (for instance, to preview without capturing). **IAMStreamControl** enables you to switch between preview and capture without rebuilding a different graph. See the source code for the VidCap Sample (Video Capture Filter) sample for details.

Time Stamping

When you send captured audio samples, the starting time stamp for each group equals the start time of the graph's clock when the first sample in the packet was captured. The ending time stamp equals the start time plus the duration that the audio packet represents. If your audio capture filter is not providing the clock, the time stamps won't match up exactly (where the end of one package is the same as the beginning time stamp of the next package), but that's okay. See Write a Video Capture Filter and the source code for the VidCap Sample (Video Capture Filter) sample for time stamping examples.

You should also set the media time of the sample you deliver, as well as the regular time stamp. The media time is the sample number in the packet. So if you are sending one-second packets of 44.1 kilohertz (kHz) audio, you would set media time values of (0, 44100) (44100, 88200), and so on. This enables the downstream filters to know if any audio samples were dropped, even when the regular time stamps are a little random because the clock being used is not the audio digitizing clock.

One other thing: If the filter graph is in a running state, and then paused, and then run again, you must not produce a sample with a time stamp less than the last one you produced before pausing. Time stamps can never go back in time, not even back to before a pause occurred.

Necessary Interfaces

Read about the following interfaces and consider implementing them. You should implement these interfaces to provide functionality that applications might rely on, so these interfaces are strongly recommended.

- Implement IAMDroppedFrames on your filter or on each output pin that sends data.
- Implement IAMStreamConfig on each output pin that sends data.
- Implement IAMStreamControl on each output pin that sends data.
- Implement IAMAudioInputMixer on your filter and on each input pin.

Write a Transform Filter in C/C++

A transform filter takes media input and alters it in some way. When you design a transform filter, your filter class derives from one of the transform base classes, CTransformFilter, CTransInPlaceFilter, or CVideoTransformFilter, or from the more generic CBaseFilter class. Which base class you choose depends on whether your filter must copy media samples or can transform them in place. See Determine Which Base Classes to Use for more information.

The filter graph manager can use the functions of the base classes your filter derives from to fit your filter into the filter graph and automatically create the connections between your filters. The filter mapper uses your filter's registry information to configure the filter graph.

For the simplest transform filter (for example, one that has only one input pin and one output pin), you can derive your filter class from CTransformFilter and override only the **Transform** and **CheckInputType** functions. If you need custom features, you can override additional functions to create your own connections, pins, and other filter features and capabilities. See Override the Base Class Member Functions for more information. You can also derive your filter class from CBaseFilter and override its methods.

This section discusses how to:

- Define and Instantiate Your Filter Class
- Override CheckInputType (does not apply to filter classes derived from CBaseFilter)
- Override the Transform Function(does not apply to filter classes derived from CBaseFilter)

- Access Additional Interfaces
- Create Registry Information

Every transform filter must implement code to perform all the preceding steps except access additional interfaces.

For background information about transform filters, see:

- Determine Which Base Classes to Use
- Override the Base Class Member Functions

For information on building a filter, see Build a Filter or Application with Visual C++ 5.x. For information on registering a filter or making it self-registering, see Register DirectShow Objects.

Define and Instantiate Your Filter Class

The following steps show you how to define and instantiate your filter class.

1. Determine the base classes from which to derive your filter class (and pin classes, if necessary). Typically, your transform filter class derives from the CTransformFilter, CTransInPlaceFilter, or CVideoTransformFilter transform base classes, or from the more generic CBaseFilter class. If you want to transform video media (especially AVI data), derive from **CVideoTransformFilter**. If your filter must copy the input media samples, derive from **CTransformFilter**. If you filter can transform the sampled media in place, derive from **CTransInPlaceFilter**. If you don't want the simple transform filter support provided in the transform base classes, but prefer to implement your own member functions, derive from **CBaseFilter**. See Determine Which Base Classes to Use for more information.

   In the following example, the filter class derives from CTransInPlaceFilter.

   ```
   class CMyFilter  : public CTransInPlaceFilter
   ```

2. Implement the IUnknown interface for your object.

   In the public section of your filter class definition, create an instance of CUnknown, and then call the DECLARE_IUNKNOWN macro.

   ```
   public:
           static CUnknown *WINAPI CreateInstance(LPUNKNOWN punk, HRESUL
   *phr);
           DECLARE_IUNKNOWN;
   ```

3. Define your constructor. Also, define your Transform and CheckInputType functions (this does not apply if your filter class is derived from CBaseFilter).

   In the private section of your filter class definition, define your constructor by calling the constructor of the transform filter class you derived from, and then add code to perform the transform and check the input type. For example:

```
//Define your constructor by calling the constructor of
//CTransInPlaceFilter
CMyFilter(TCHAR *tszName, LPUNKNOWN punk, HRESULT *phr)
: CTransInPlaceFilter (tszName, punk, CLSID_MyFilter, phr)
{ }

//Add the transform code
HRESULT Transform(IMediaSample *pSample){
//Transform code here
}

//Add code to check the input type
HRESULT CheckInputType(const CMediaType* mtIn) {
//Input checking code here
}
```

4. Implement **CreateInstance** for your filter object. Typically, **CreateInstance** calls the constructor of your filter class. For example:

```
CUnknown * WINAPI CMyFilter::CreateInstance(LPUNKNOWN punk, HRESULT *]
CMyFilter *pNewObject = new CMyFilter(NAME("Description of My Filter"
if (pNewObject == NULL) {
        *phr = E_OUTOFMEMORY;
}
return pNewObject;
}
```

5. Declare a global array g_Templates of CFactoryTemplate objects to inform the default class factory code how to access the **CreateInstance** function:

```
      CFactoryTemplate g_Templates[]=
  {     { L"My Filter"
              , &CLSID_MyFilter
              , CMyFilter::CreateInstance //Function called by class factor
              , NULL
              , &sudMyFilter } //Address of the AMOVIESETUP_FILTER structur
                                  //or NULL if no structure exists
      };
  int g_cTemplates = sizeof(g_Templates)/sizeof(g_Templates[0]);
```

The g_cTemplates variable defines the number of class factory templates for the filter. Each of these templates provides a link between COM and the filter and are used to create the base object for the filter. At a minimum, the filter has one template that provides the address of its own **CreateInstance** function, which, when called, creates the base object.

You can add additional parameters to the CFactoryTemplate templates to add property pages. See the Gargle sample for example code. See Register DirectShow Objects for information about using **CFactoryTemplate** in registration.

6. Generate a GUID for your filter object.

For information about generating GUIDs in general, see "GUID Creation and Optimizations" and "The uuidgen Utility" in the Platform SDK.

To generate a GUID in Microsoft® Visual C++® 5.*x*, choose **Create GUID** from the **Tools** menu. By default, the **GUID** is in DEFINE_GUID format, which is the format you want. Click the **Copy** button. Put the cursor in your source file beneath the include statements, and choose **Paste** from the **Edit** menu. The inserted code will look like the following example, except that it will have its own unique number and CLSID. Insert the code before your class definition in the header file or main file.

```
// {3FA5D260-AF2F-11d0-AE9C-00A0C91F0841}
DEFINE_GUID(CLSID_MyFilter,
0x3fa5d260, 0xaf2f, 0x11d0, 0xae, 0x9c, 0x0, 0xa0, 0xc9, 0x1f, 0x8, 0:
```

Override CheckInputType

You must override the **CheckInputType** function to determine if the proposed input to your filter is valid. (This does not apply to filter classes derived from CBaseFilter.) Your implementation should return an error for media types it can't support. The media types your filter supports are listed in the AMOVIESETUP_MEDIATYPE structure. For example:

```
HRESULT CMyFilter::CheckInputType(const CMediaType *pmt)
{
if (pmt->majortype != MEDIATYPE_Video) {
        return S_FALSE;
}
        else return S_OK;
}
```

Override the Transform Function

To perform the desired transformation on your input media, your must override the **Transform** function of your transform base class, or implement your own transformation functions. (This does not apply to filter classes derived from CBaseFilter.) Examples of transformations are MPEG audio/video decoders (see the MPGAudio and MPGVideo samples), visual effects (see the Contrast and EzRGB24 samples), and audio effects (see the Gargle sample).

For example, consider the following code from the Contrast sample. You override the CContrast::Transform function as follows:

```
HRESULT CContrast::Transform(IMediaSample *pIn, IMediaSample *pOut)
{
        HRESULT hr = Copy(pIn, pOut);
if (FAILED(hr)) {
        return hr;
}
        return Transform(pOut);

}
```

The first CContrast::Transform function copies the media data, and then passes the copy (pointed to by the *pOut* parameter) to a second Transform function. The first Transform function in the Contrast sample is an overloaded function, and the second form of the Transform function performs an in-place transform on the copy of the input media, as shown in the following code fragment.

```
HRESULT CContrast::Transform(IMediaSample *pMediaSample)
```

```
{
signed char ContrastLevel;
ContrastLevel = m_ContrastLevel;
AM_MEDIA_TYPE *pAdjustedType = NULL;

pMediaSample->GetMediaType(&pAdjustedType);
HRESULT hr = Transform(&AdjustedType, ContrastLevel);
pMediaSample->SetMediaType(&AdjustedType);
return NOERROR;
}
```

Note that the second form of the overloaded Transform function calls a third form of the overloaded Transform function.

Access Additional Interfaces

If your filter implements any interfaces that aren't implemented in the base classes, you must override the **NonDelegatingQueryInterface** function and return pointers to the implemented interfaces.

1. In the public section of your filter class definition, declare **NonDelegatingQueryInterface**:

   ```
   STDMETHODIMP NonDelegatingQueryInterface(REFIID riid, void ** ppv);
   ```

2. In the implementation section of your class, implement the **NonDelegatingQueryInterface** function. For example:

   ```
   //Reveal persistent stream and property pages
   STDMETHODIMP CMyFilter::NonDelegatingQueryInterface(REFIID riid, void
   {
   if (riid == IID_IPersistStream) {
   AddRef( );   // Add a reference count. Be sure to release when done.
   *ppv = (void *) (IPersistStream *) this;
   return NOERROR;
   }
   else if (riid == IID_ISpecifyPropertyPages) {
           return GetInterface((ISpecifyPropertyPages *) this, ppv);
   }
   else {
   return CTransInPlaceFilter::NonDelegatingQueryInterface(riid, ppv);
   }
   }
   ```

Create Registry Information

The filter graph manager uses your filter's registry entries to configure your filter and its connections. You provide your filter's registry information in the AMOVIESETUP_MEDIATYPE, AMOVIESETUP_PIN, and AMOVIESETUP_FILTER structures. Typically, these structures are at the beginning of your filter implementation code. See Register DirectShow Objects for more information about using these structures.

Perform the following steps to provide the three structures you need for filter registration.

1. Provide the AMOVIESETUP_MEDIATYPEstructure. This structure holds registry information about the media types your filter supports. For example:

```
        const AMOVIESETUP_MEDIATYPE sudPinTypes =
                        { &MEDIATYPE_Video              // MajorType
                        , &MEDIASUBTYPE_NULL}  ;        // MinorType
```

The possible major types are MEDIATYPE_Stream, MEDIATYPE_Video, and MEDIATYPE_Audio.

2. Provide the AMOVIESETUP_PIN structure. This structure holds registry information about the pins your filter supports.
3. Provide the AMOVIESETUP_FILTER structure. This structure holds registry information about your filter object: its CLSID, description, number of pins, the pin structure's name, and your filter's merit. The *merit* controls the order in which the filter graph manager accesses your filter. Possible merit values are MERIT_PREFERRED, MERIT_NORMAL, MERIT_UNLIKELY, and MERIT_DO_NOT_USE. See IFilterMapper::RegisterFilter for a description of merit values. The following code shows an example of an **AMOVIESETUP_FILTER** structure.

```
        const AMOVIESETUP_FILTER
        sudMyFilter = { &CLSID_MyFilter                 // clsID
                , L"My Filter Description"              // strName
                , MERIT_UNLIKELY                                // dwMerit
                , 2                                             // nPins
                , sudpPins };                           // lpPin
```

Stream Architecture

This section describes the DirectShow stream architecture and connection model. Topics include connecting filters, using pins, and negotiating data types.

About Stream Architecture

Stream architecture defines objects and interfaces that exchange streams of time-based data. In particular, it defines interfaces for the following requirements.

- Connecting filters to other filters.
- Negotiating data types.
- Transporting data between filters.
- Synchronizing presentation of data.
- Graceful degradation of rendering in cases of insufficient resources (that is, *quality-control management*).

  See Filters and Pins for more information.

Previous Home Topic Contents Index Next

Connection Model

This article provides an overview of the filter connection architecture in a Microsoft® DirectShow™ filter graph by examining the behavior of the base classes that implement connection. Because filters connect to other filters using pins, the architecture describes pin connection. Consequently, the CBasePin, CBaseOutputPin, and CBaseInputPin base classes and the IPin interface are discussed. This article describes the connection process and the default functionality built into these classes.

Contents of this article:

Connection Process

When building a filter graph, the filter graph manager connects pins between filters. It also selects filters based on the media type in the file it has been given to render or selects a predetermined configuration for the filter graph it is assembling. The filter graph manager can be asked specifically to add a filter by using the IFilterGraph::AddFilter method. The filter graph manager calls the IBaseFilter::JoinFilterGraph method on the filter to notify it that it has joined the filter graph. The added filter can then be connected like any other filter. When connecting filters, the filter graph manager requests the filters to enumerate their pins and then, for each connection required, requests that an output pin connect to an input pin.

The base classes handle much of the connection mechanism. However, it is important to understand the connection process when writing a filter so that you can identify what to override and what is expected of your filter. Before two connected filters are prepared to pass media between them, the following connection and negotiation processes must occur in this order.

1. The initial pin connection occurs.
2. The output pin of one filter retrieves interfaces from the connected input pin.
3. Both pins negotiate for a common media type.
4. Both pins negotiate for an appropriate transport to pass the media.

In the first step, the filter graph manager informs the output pin of one filter to connect to a specified input pin of another filter. This results in an exchange of IPin interface pointers. Filters should never connect to other filters by themselves. The filter graph manager must always be the agent that initiates a connection, because deadlocks can occur otherwise. A filter or an application can instruct the filter graph manager to connect two pins (through the IGraphBuilder::Connect or IFilterGraph::ConnectDirect method), or the filter graph manager can determine to connect filters when rendering a filter by using the IGraphBuilder::Render or IGraphBuilder::RenderFile method.

In the second step, the output pin may request the IMemInputPin interface from the connected input pin. This is in preparation for the fourth step, where the output pin will use **IMemInputPin** to retrieve a memory allocator from the input pin. If the output pin already has a memory allocator (or some other transport in the case of hardware filters), it can skip this step or can request some other interface in a proprietary design.

In the third step, media types are tried until one is found that is acceptable to both pins or the pins run out of types to try (in which case the connection fails). First, the output pin asks the connected input pin to propose its list of media types. If none of these are acceptable to the output pin, the output pin proposes its own types.

In the fourth step, the output pin asks the connected input pin for an allocator interface. The output pin then either accepts the allocator, or proposes its own allocator and notifies the input pin of the selection. The output pin makes the final determination.

How the Base Classes Implement Connection

The CBasePin class and its derived base classes, CBaseOutputPin and CBaseInputPin, implement most of the mechanism for the most common connection scenarios, much of which can be overridden by the derived filter class for more control of the process.

The connection procedure relies on the implementation of four interfaces:

1. IPin, which is implemented by the CBasePin class and inherited by the CBaseInputPin and CBaseOutputPin classes.
2. IEnumMediaTypes, which is implemented by the CEnumMediaTypes class and passed out by the IPin::EnumMediaTypes method.
3. IMemInputPin, which is implemented by the CBaseInputPin class.
4. IMemAllocator, which is implemented by the CBaseAllocator class and passed out by the IMemInputPin::GetAllocator method.

The IMemInputPin and IMemAllocator interfaces are necessary only if the filter belonging to the connecting input pin (called the *downstream filter*) is expected to provide a shared memory allocator for transporting samples between the pins. However, the base class implementation in CBaseInputPin assumes this condition and implements **IMemInputPin** to provide an allocator object to a connected output pin that requests it.

In the connection scenario of the default base class, the pin classes derived from CBaseInputPin and CBaseOutputPin need only to override and implement a few member functions and can let the base classes do the remaining work. Base classes derived from these classes, such as CTransformInputPin and CTransformOutputPin, do much of the required implementation to provide a default connection scheme.

Pin classes derived from CBaseInputPin and CBaseOutputPin need only to override the following member functions to enable pin connection.

- CBasePin::CheckMediaType, which is called for every media type proposed by the media type enumerator. The overriding member function must accept or reject the proposed media type.
- CBasePin::GetMediaType, which is called by the media type of the output pin enumerator to suggest media types already agreed on by the input pin for transform filters. This member function also presents the type of media a source filter will produce.

Additionally, the output pin derived from the CBaseOutputPin class must override the CBaseOutputPin::DecideBufferSize member function. This is called by the base classes to let the output pin inform any acquired allocator of the size and type of media samples that the output pin will provide. This is done by the output pin of the filter because the derived filter class should know the type and size of the data it will send to the input pin of the connected filter.

To see the context of these overriding functions, it is helpful to step through the execution of the connection code in the class library. All connection takes place in the scope of one CBasePin::Connect member function.

The Filter Graph Manager Starts the Connection

The connection starts when the filter graph manager calls the IPin::Connect method on the output pin, passing it a pointer to the input pin to which it is connecting. The filter graph manager has previously retrieved pointers to the IPin interfaces of both filters, for example, by calling the IBaseFilter::EnumPins method on each connecting filter. The EnumPins method creates a CEnumPins object to enumerate the pins, which the enumerator does by repeatedly calling the CBaseFilter::GetPin member function of the derived filter, which the derived filter must implement.

The CBasePin::Connect implementation of IPin::Connect does much of the work in this case. It calls the following functions.

- CheckConnect, which is overridden by CBaseOutputPin.
- AgreeMediaType, which is implemented by CBasePin.

The CBasePin::CheckConnect implementation simply determines that the pin directions are different. The overriding CBaseOutputPin::CheckConnect member function calls the IUnknown::QueryInterface method of the connected input pin to retrieve a pointer to the IMemInputPin interface of that pin. This will be used later in the connection process to request an allocator from the connected input pin. (Your derived class can override **CBaseOutputPin::CheckConnect** and omit retrieving the **IMemInputPin** interface if the output pin already has an allocator; for example, it might want to use the allocator from an upstream filter to eliminate copying.)

Negotiating Media Types with CBasePin::AgreeMediaType

The CBasePin::AgreeMediaType member function is called next and attempts to negotiate a media type that both pins agree on. It does this by trying to find a media type presented by the connected input pin with which the output pin agrees. If that fails, it tries to find a media type preferred by the output pin that the connected input pin agrees with.

CBasePin::AgreeMediaType calls the following member functions and methods.

- IPin::EnumMediaTypes on the connected pin.
- CBasePin::TryMediaTypes in the derived output pin class.

The IPin::EnumMediaTypes method of the connected input pin is called to return a media type enumerator (IEnumMediaTypes). This allows the output pin to examine the list of preferred media types belonging to the input pin.

The IEnumMediaTypes::Next method of the enumerator calls the GetMediaType member function of the derived input pin to retrieve each media type. If **GetMediaType** is not implemented, the base class implementation returns an error but this does not necessarily break the connection. (Pins are not required to have a preferred media type if one pin or the other can propose a type that they both accept. If neither pin can propose types, the connection will fail.)

Determining a Media Type with CBasePin::TryMediaTypes

CBasePin::AgreeMediaType calls CBasePin::TryMediaTypes next. The TryMediaTypes member function cycles through the preferred media types of the connected input pin and calls the CBasePin::CheckMediaType member function of the derived output pin class for each one it finds. CheckMediaType must be implemented by your derived output pin class. If **CheckMediaType** accepts the media type, the IPin::ReceiveConnection method of the connected input pin is called with the media type to determine if the connected input pin accepts this media type. If so, **TryMediaTypes** calls the CBaseOutputPin::CompleteConnect member function to finish the connection to the input pin.

If the input pin has no media types that the output type can use, CBasePin::AgreeMediaType repeats the entire process, using the enumerator for the media types of the output pin. (That is, it gets its own enumerator and calls TryMediaTypes with each of its preferred media types.) Again, the enumerator calls GetMediaType for each media type in the list. In this case, **GetMediaType** should be implemented to provide a media type. If the filter is a source filter, it will have a definite media type to export. If the filter is a transform filter, the media type will be established between the filter's input pin and its connected pin; the transform filter should query for that media type or simply use the enumerator of the upstream filter (unless the transform filter changes the media type from input pin to output pin).

CheckMediaType is called by CBasePin::TryMediaTypes, even when TryMediaTypes enumerates the list of the preferred media types of the output pin. This is because the owning filter might be a transform-inplace filter that is simply using the media type (and enumerator) of an upstream filter; this is the point at which the filter determines if the media type is compatible. The input pin of this transform filter might defer selecting a media type when it is connected, in which case it would be up to the output pin of the transform filter to ensure the media type is compatible with its transform.

If a media type can be established, TryMediaTypes eventually calls the CBaseOutputPin::CompleteConnect member function to negotiate a memory allocator.

First, the CBaseOutputPin::CompleteConnect member function calls the CBaseOutputPin::DecideAllocator member function. This member function negotiates a shared memory allocator with the input pin. It does this by first calling the IMemInputPin::GetAllocator method of the connected input pin, which retrieves a pointer to an IMemAllocator interface provided by the input pin.

Then, CompleteConnect calls the pure virtual CBaseOutputPin::DecideBufferSize member function, which your derived output pin class must override and implement because only the derived class can determine the required buffer size for its media type.

Finally, CompleteConnect calls the IMemInputPin::NotifyAllocator method of the connected pin to inform the input pin of the allocator to use and to provide a pointer to it. The input pin can reject this allocator, in which case the output pin can retry with a different allocator or fail the connection. If your derived class is not using the allocator of the connected input pin, override CBaseOutputPin::DecideAllocator in your derived class to call the NotifyAllocator member function with an allocator.

When a Reconnection Should Occur

Reconnection is always performed through the IFilterGraph interface on the filter graph manager. Reconnect by calling the IFilterGraph2::ReconnectEx method or the IFilterGraph::Reconnect method, both of which pass the IPin interfaces of the two pins to be reconnected. The ReconnectEx method specifies a media type and thus removes the burden of remembering what type to reconnect with from the pins, which makes the reconnection more likely to succeed.

Filters are typically connected with the upstream filter first and the downstream filter second. Therefore, the filter negotiates the connection on its input pin before information is available about the filter being connected to its output pin. When the output pin of the filter connects, it may become clear that the media type or allocator that was established for the input pin of the filter are not appropriate. In this case, the input connection can be broken and reconnected.

For example, consider the following connection scenario. An audio effects filter (for example, a reverberation effect) is inserted between an MPEG-audio decompressor filter and another audio effects filter. During the upstream connection to the decompressor filter, a media type is chosen—for example, 22.05 kHz, 16-bit mono. However, in this scenario, when the reverberation filter connects its output pin, the downstream filter will accept only an 11.025 kHz, 16-bit mono media type. Therefore, after connecting with the downstream filter, the reverberation effects filter must then reconnect with the upstream filter and negotiate for an 11.025 kHz media type.

But media types are not the only reason for reconnection. In many cases, the filter is a *transform-inplace filter*; that is, a filter that does not require that it either alters the media type or copies the data. Such a filter can be designed to use an allocator of some other filter (upstream or downstream), and likewise use the media type of another filter. That is, the filter is doing its transform "in place" in the buffer of another filter (for example, in the file buffer of the source filter or the video buffer of the rendering filter).

The general rule is that filters of this type should offer the allocator of the downstream filter to the upstream filter, once the allocator has been established for the output pin. This requires a reconnection of the input pin so that, when the input pin is asked for an allocator (in IMemInputPin::GetAllocator) by the upstream output pin, it can offer the allocator retrieved from the downstream filter by the output pin of the transform filter. Therefore, in-place transforms always reconnect.

There are a couple of important rules to follow when requesting a reconnection.

First, a filter must never request a reconnection unless it is certain that the reconnection will succeed. If the reconnection fails, it causes an asynchronous error in the filter graph for which there is no obvious cleanup. Any error that occurs (for example, from incompatible media types) should occur when the pins are connected the first time, when there are ample retry options available at more than one level (by the filter graph manager or the application at least).

Second, a filter should request a reconnection on the same thread as the call to IPin::Connect. For example, the following scenario attempts reconnection on a separate thread and will cause problems.

1. The filter graph manager calls Connect on a pin.
2. The filter pin carries out the Connect method and creates a thread, which starts to determine whether everything is okay for the connection.
3. Connect returns to the filter graph manager.
4. The filter graph manager returns to the application.
5. The application calls the IMediaControl::Run method of the filter graph manager to start the filter graph, and the filters start running.
6. The thread from the initial connection calls the IFilterGraph2::ReconnectEx or IFilterGraph::Reconnect method and the filter graph manager attempts to carry out reconnection.
7. Failure occurs because the filters cannot reconnect while in the running state.

The filter graph has code to prevent this failure as long as the IFilterGraph2::ReconnectEx or IFilterGraph::Reconnect method takes effect while the filter graph is still processing the IGraphBuilder::Connect method. Calling the filter graph to reconnect before returning from the IPin::Connect method is the best way to ensure this problem does not occur. The best way to achieve this is to perform all of this on the same thread.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

Plug-in Distributors

This article describes the plug-in distributor architecture and provides some rules assumed by the default Microsoft® DirectShow™ control distributors.

Contents of this article:

- Plug-in Distributors and Extensibility
- Control Distributors

Plug-in Distributors and Extensibility

The filter graph manager exports control interfaces; it also distributes the actions of interface methods to the appropriate filters. For example, the IMediaControl::Run method on the filter graph manager is called by an application to run the filter graph; this command is distributed to the IMediaFilter::Run method of each filter method by the filter graph manager. This distribution allows applications to have a single point of control to perform the basic operations.

To allow the filter graph manager to be extensible, a mechanism known as a *plug-in distributor* (PID) is used. This is a Component Object Model (COM) object that exposes a particular control interface and implements it by calling the enumerator of the filter graph manager, finding which filters expose the control interface and communicating directly with those filters. PIDs are supplied for the standard control interfaces; independent software vendors (ISVs) can replace these supplied PIDs and also add others.

When the filter graph manager is asked for an interface that it does not recognize, it searches the registry for a PID. This is an unnamed value under the following key.

```
HKEY_CLASSES_ROOT\Interface\<IID>\Distributor
```

This value provides the class identifier (CLSID) of an object that can distribute the interface identifier (IID). The filter graph manager then instantiates that object as an aggregated object, specifying the IUnknown implementation of the filter graph manager as the outer **IUnknown**, and asking for the IID. The object will then be able to use its outer **IUnknown** pointer to obtain an IFilterGraph interface. With this interface, it can enumerate the filters to implement its control interface methods and properties; it will also be able to use the IMediaControl implementation of the filter graph manager for correctly ordered and synchronized state changes (run, pause, stop, and so on).

Control Distributors

A control distributor is a PID that is used to control the data flow in the filter graph; for example, starting or stopping playback of a media stream. The standard control distributors supplied with DirectShow directly implement their distribution. These distributors make the following assumptions:

- Applications that connect filters directly without informing the filter graph manager will get unexpected results if they also use the distributors of the filter graph manager. For example, a deadlock might occur if an application calls a filter's IBaseFilter::Run method directly, because the filter graph manager contains a distributor that implements IMediaControl::Run and passes calls on to each filter's **IBaseFilter::Run** method.
- Properties that can be aggregated directly can be read and written to through the control interface, even when exposed by multiple filters. For example, duration can be reported as the longest of all individual durations, with all streams treated as running in parallel.
- Where a property is exposed by several different filters, applications will either use the filter graph manager to set and get the property or will communicate with the individual filters, but will not mix the two methods. An application that communicates with two audio renderers to reset the volume and then queries the IBasicAudio implementation of

the filter graph manager for the volume, will get undefined results. (In practice, it will probably retrieve the **Volume** property of the first audio renderer with no attempt to combine this with the other stream.) If the application sets the property through the interface of the filter graph manager, the same value will be set to all the individual filters that expose it.
- The filter graph manager will expose the IMediaControl interface (through a non-replaceable distributor) as the main application method for starting and stopping graphs. This is a slightly higher-level, more simplified interface than IMediaFilter and is suitable for Automation clients and applications. The **IMediaFilter** implementation on the filter graph manager should not be called by applications. **IMediaControl** is implemented by calling the **IMediaFilter** interface implemented by the filter graph manager and by individual filters. Individual filters expose **IMediaFilter** through the IBaseFilter interface, which inherits it.

PIDs must keep track of the filters in the filter graph. This is done by implementing the IDistributorNotify interface on the distributor. **IDistributorNotify** has the same Run, Pause, and Stop methods as IMediaFilter, all of which are called before the calling the filter. It also has a IDistributorNotify::NotifyGraphChange method, which notifies the distributor when any filters are added or removed from the filter graph, or connections change.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

DirectShow and COM

Microsoft® DirectShow™ provides a framework that simplifies the creation of Component Object Model (COM) objects. This article describes this framework and most of what you need to know about COM to create a filter or plug-in distributor using the C++ class library. The article assumes the reader is familiar with C++. An understanding of COM would be helpful, but is not essential.

Contents of this article:

COM Objects in DirectShow

DirectShow filters, the filter graph manager, plug-in distributors, and enumerators are all COM objects. A general design has been adopted for the way in which DirectShow implements COM objects. This design is available to help you implement your own filters and plug-in distributors (or any COM object).

DirectShow components are supplied as in-process servers; that is, servers that run in the same address space as your application. They are packaged in a single dynamic-link library (DLL), Quartz.dll. Use the COM framework of DirectShow to build your own in-process COM servers, which you can package in your own DLL(s).

Typically, a single C++ class implements a single COM class. The DirectShow COM framework requires that C++ classes implementing COM objects conform to a few simple rules. One of these rules is that the developer provides a class factory template for each such class. The *class factory template* contains information about the class that is vital to the framework. Class factory templates are defined in the DLL using two global variables (g_Templates and g_cTemplates) as shown in the following example.

```
CFactoryTemplate g_Templates[]=
{   {L"My class name", &CLSID_MyClass,   CMyClass::CreateInstance,  CMyClass::Init}
{L"My class name2", &CLSID_MyClass2,   CMyClass2::CreateInstance}
};
int g_cTemplates = sizeof(g_Templates)/sizeof(g_Templates[0]);
```

The names and types of these variables must be as they appear in the previous example. Because any DLL might contain several COM classes, each of which will require a class factory template, the factory templates are defined in an array and the number of elements in the array is recorded in another variable. Each element of the array contains the following fields.

- A textual description of the class (using wide characters, therefore the "L" prefix).
- A pointer to the class identifier of the class (CLSID).
- A pointer to a static method of the class that can create instances of the class (CFactoryTemplate::CreateInstance).
- A pointer to a static method of the class. This method is called when the DLL is loading or unloading and can perform one-time initialization and termination. If this method is not required, this can be omitted, will default to NULL, and will be ignored.
- A pointer to an AMOVIESETUP_FILTER structure. This is required when using filter self-registration services.

The DirectShow COM framework uses the information in these class factory templates to create instances of the specific class, and to register and unregister the COM classes.

The following example demonstrates a simple C++ class implementing a COM class using the DirectShow framework.

```
class CMyClass : public IMyInterface, public CUnknown
{
private:
        /* private attributes */
protected:
        ~CMyClass()
        { /* release private attributes */ }
        CMyClass(TCHAR *pName, LPUNKNOWN pUnk, HRESULT *phr)
        : CUnknown( pName, pUnk, phr )
        { /* set up private attributes */ }
public:
        DECLARE_IUNKNOWN
static CUnknown *CreateInstance(LPUNKNOWN pUnk, HRESULT *phr)
{
CUnknown * result = 0;
result = new CMyClass( NAME("CMyClass"), pUnk, phr );
if ( !result ) *phr = E_OUTOFMEMORY;
return result;
}

STDMETHODIMP NonDelegatingQueryInterface(REFIID iid, void ** ppv)
{
if ( iid == IID_IMyInterface )
{
return GetInterface(static_cast<IMyInterface *>(this), ppv );
}
           else
                {
                        return CUnknown::NonDelegatingQueryInterface(iid, ppv);
                }
}

        /* My interface methods */
};
```

This is not a typical filter example, because filters will normally derive from more specialized base classes than CUnknown. However, because all base filter classes eventually derive from **CUnknown**, this example demonstrates what is essential in a more generic manner. (The example is probably more typical for a DirectShow plug-in distributor (PID), which extends the functionality of the filter graph manager, or for a framework for implementing an arbitrary COM object.)

In this example, the NonDelegatingQueryInterface method is implemented. The more specialized filter base classes that derive from **CUnknown** are responsible for implementing **NonDelegatingQueryInterface** for the required interfaces; this is only necessary in the derived filter class if it adds some interfaces that are not in the base classes. In this case, it adds its own interface, IMyInterface.

INonDelegatingUnknown::NonDelegatingQueryInterface is a method that allows other objects to access interfaces on the COM object. All COM objects support IUnknown::QueryInterface to do this, and the DirectShow class library supplies the DECLARE_IUNKNOWN macro to enable the IUnknown interface. The DirectShow framework goes one step further and makes it easy to aggregate objects (make them part of a larger COM object) by implementing an INonDelegatingUnknown interface. Even if your object is not aggregated, it uses the **INonDelegatingUnknown** interface, which is mapped to the **IUnknown** interface by the base classes.

Although aggregation is handled for all objects by the DirectShow class framework, it is typically not used by filters in current DirectShow filter graphs. Plug-in distributors do, however, require aggregation (as is described later in this article), and future filter graphs might incorporate filter objects that are composed of collections of aggregated filters.

With this in mind, it might be helpful to explore more of the details of the previous example. First, a brief review of some COM basics might be helpful. COM objects are created by their class factories, are reference counted during their lifetimes, and self-destruct when their reference counts drop to zero. COM objects can be created in isolation, or can be aggregated with an already existing COM object. In this second case, the existing object (referred to as the outer object) maintains the reference count. The created object (referred to as the inner object) is not reference counted, but will be destroyed by the outer object during the destruction of the outer object. The application cannot directly manipulate COM objects; an application can only invoke the methods, which the object chooses to expose through its interfaces. Typically, COM objects make several interfaces available. All COM objects must support the IUnknown interface.

All classes using the DirectShow framework must inherit from CUnknown either directly (as in the previous example) or indirectly, through one of the other supplied base classes. **CUnknown**, with the DECLARE_IUNKNOWN macro and the NonDelegatingQueryInterface method, provide the IUnknown interface with the required reference counting and support for COM aggregation.

NonDelegatingQueryInterface is a method on INonDelegatingUnknown, which is supported by CUnknown. **NonDelegatingQueryInterface** is overridden in derived classes that support new interfaces, such as IMyInterface in the previous example. The method should check for all the interfaces known to be implemented on the object and return appropriate pointers to these interfaces. Requests for unrecognized interfaces should be passed to the **NonDelegatingQueryInterface** of **CUnknown**. The call to the GetInterface method (of **CUnknown**) copies the interface into the *ppv* parameter and ensures that the correct reference count is incremented.

The methods in INonDelegatingUnknown mirror those in IUnknown. For more information about CUnknown, the **INonDelegatingUnknown** interface, and the NonDelegatingQueryInterface method, see the **CUnknown** section in the reference material. **INonDelegatingUnknown** is defined in Combase.h; **CUnknown** is implemented in Combase.cpp.

When an instance of the class is required, the framework, using the information in the class factory template, calls the derived class's **CreateInstance** member function. The framework passes a pointer to an outer unknown (if the object will be part of an aggregate object) through the *pUnk*parameter, and passes a pointer to an HRESULT value through the *phr* parameter. The constructor of an inherited class can set this value if an error occurs. The *phr* parameter should not be initialized; this is the calling application's responsibility. The **CreateInstance** member function constructs an instance of the class by calling the constructor. The name passed to the constructor is wrapped with the NAME macro supplied by DirectShow. When building debugging versions, **NAME** passes the textual name on to the constructor. When building nondebugging versions, **NAME** results in a null pointer, thus saving space in versions that are not for debugging purposes.

The class constructor and destructor are declared protected. This prohibits the creation of the object using C++ language constructs. Instances of this class can be created only by calling the **CreateInstance** member function.

The class constructor needs to construct the inherited CUnknown. The *pName* parameter points to a string that is available for debugging purposes. It is vital that the string referenced by *pName* is in static storage, because the constructor for **CUnknown** will not copy it.

Reviewing the Instantiation Process

It might be helpful at this point to consider the normal process of instantiating a COM object, and examine how the DirectShow COM framework supports this process. First, a look at the entry points required of an in-process server DLL (such as a filter or plug-in distributor) is in order.

In-process server DLLs must export certain standard functions so that COM can interact with them. The DirectShow framework provides these functions for you. The module definition file for the DLL must list these functions in its EXPORTS section, and link to Strmbase.lib. The functions are: DllGetClassObject and DllCanUnloadNow. (The source code for these functions is supplied in Dllentry.cpp.)

A DirectShow object can define DLL entry points that facilitate the automatic registration of COM classes. These entry points are DllRegisterServer and DllUnregisterServer. Although the framework does not directly provide these entry points, it does provide a function, called AMovieDllRegisterServer2, that can implement these entry point functions. These functions take care of registering and unregistering all COM objects for which you have provided class factory templates in the g_Templates array. You can add a DllRegisterServer function to your module that simply calls **AMovieDllRegisterServer2**, or you could do the same for DllUnregisterServer. For more information on self-registering DirectShow COM objects, see Register DirectShow Objects.

Registry entries are required to link the class identifier (CLSID) of the COM object to the DLL in which the class is implemented. The framework provides entry points in the DLL that support the automatic registration of class identifiers in the registry, using the information provided in the class factory templates.

Following are the steps that occur during initialization, which require the entry points mentioned previously.

1. When the DLL is loaded, the DllMain entry point is called to perform any initialization. The framework provides this function. During its execution, any initialization routines referenced in the class factory templates will be called.
2. When an application calls CoCreateInstance or CoGetClassObject, COM calls the DllGetClassObject function in the appropriate DLL to obtain a pointer to a class factory that can instantiate objects of the CLSID requested by the application. The framework supplies this function. Using the information in the class factory template, the framework creates a class factory. (If the requested CLSID cannot be found in the array of class factory templates, an error is returned to the application.)
3. The class factory is called to instantiate an object that supports the interface identifier (IID) requested by the application. At this point, the class factory will call the static method referenced in the class factory template.
4. During the DLL's lifetime, the QueryInterface method might be called on the IUnknown interface of the object (or owning object if aggregated), requesting some interface on that object. By deriving the object class from CUnknown, overriding NonDelegatingQueryInterface, and using the DEFINE_IUNKNOWN macro to declare the **IUnknown** interface, both COM aggregation and reference counting are addressed.
5. During the life of the DLL, DllCanUnloadNow might be called to see if it is safe to unload the DLL. Typically, this returns S_FALSE if any class factory is locked, or if any of the objects that have been created still exist. The framework implements **DllCanUnloadNow**.

Creating Filters

When creating filters, you can take advantage of one of the richer classes that DirectShow provides, such as CTransformFilter or CBaseRenderer, instead of deriving from CUnknown. These supplied classes are derived from **CUnknown**, but provide additional functionality specific to various types of filters. However, building filters also requires an understanding of the DirectShow connection model (see Connection Model) and the pin classes. For more information about creating filters, see Creating a Transform Filter.

Creating Plug-in Distributors

The filter graph manager can perform operations at a high level, treating the filter graph as a single entity. These operations can be distributed across an entire filter graph, or perhaps confined to just a single filter in the filter graph. The filter graph manager, of itself, only exposes a few interfaces. A feature called a plug-in distributor allows the filter graph manager to be extended with additional interfaces. When the filter graph manager receives a request for an interface which it does not support, it tries to find a plug-in distributor (PID) that does support it. If it succeeds in finding such a PID, then that PID is instantiated as an aggregate object within the filter graph manager. By doing so, the filter graph manager appears to support many more interfaces. Plug-in distributors are aggregated with the filter graph manager, but all the aggregation logic is provided by CUnknown, allowing you to concentrate on the PID logic.

A PID is designed to be aggregated into a filter graph manager; it will call on the services of its owning filter graph manager. Because the PID is unlikely to function correctly without an owning filter graph, it checks for an outer unknown in the constructor of the PID. To make this determination, add the following line to the body of the constructor illustrated in the previous example.

```
if (!pUnk) *phr = VFW_E_NEED_OWNER;
```

To be even more defensive against being used without an owner, the PID could also request an IFilterGraph or IGraphBuilder interface from the outer unknown during construction, because these interfaces are known to be only on the filter graph manager.

If the PID obtains any interface pointers from the filter graph manager, the pointers should be released immediately. Because the PID is an aggregate object, its lifetime is within the lifetime of its containing object, the filter graph manager, so there is no need to maintain a lock on it. Furthermore, maintaining a lock introduces a circular reference count that would not allow the destruction of the filter graph manager.

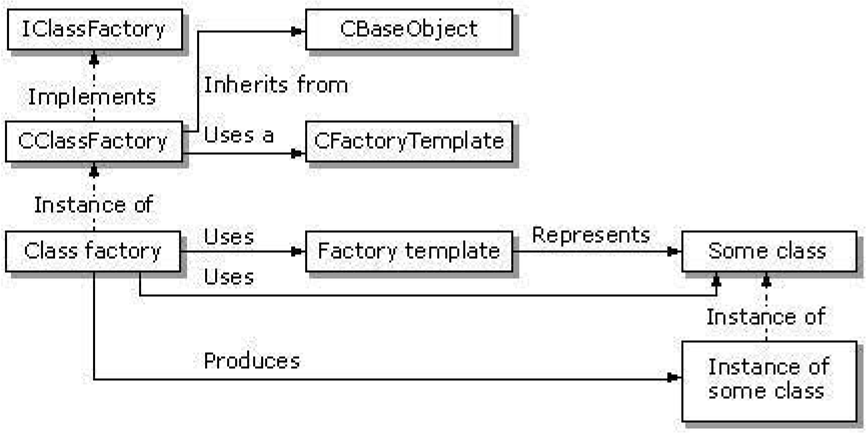
Implementing the Class Factory

The concept of a class factory is not specific to DirectShow; it is a common design that appears when the underlying type of the object being created is not known to the client that requests its creation. With COM objects, clients request interface pointers but know little about the underlying objects that implement that interface.

In C++, there are two means of implementing a class factory. One is to implement it as a genuine class, the other is to implement it as a static method on the class that the class factory will manage. The first method provides better separation of responsibilities and data hiding, and is the approach adopted by COM. The second method allows for a simpler implementation.

The DirectShow COM framework provides the best of both worlds. It exposes a genuine COM factory class to its clients while allowing the developer to implement the body of the class factory as a static method of your class. The bridge between these two approaches is two global variables, g_Templates and g_cTemplates, which were described previously.

The DirectShow framework defines two classes for implementing the class factory: CFactoryTemplate and CClassFactory. A **CFactoryTemplate** object holds information regarding a specific class, including a pointer to the static factory method of the class. When **CClassFactory** is instantiated, it must be given reference to a **CFactoryTemplate** instance. The **CClassFactory** instance will then act as a class factory for the class described in its associated **CFactoryTemplate** instance. The following illustration demonstrates the relationship between these classes, their instances, and the objects they create.



The DirectShow SDK includes a module, DllEntry.cpp, which provides the DllGetClassObject function. This function uses the process described previously to create a class factory that can produce instances of a class.

Using an Object-Oriented Model

All components of the DirectShow filter graph architecture are implemented as COM objects. This includes the filters through which data is passed, and filter components that serve as a connection between filters or allocate memory. Each object implements one or more interfaces, each of which contains a predefined set of functions, called *methods*. An application calls a method, or other component objects, to communicate with the object exposing the interface. For example, the application calls methods on the IMediaControl interface on the object of the filter graph manager, such as the Run method, which starts a media stream. The filter graph manager, in turn, calls the Run method on the IBaseFilter interface exposed by each of the filters.

Filter graph architecture uses COM interfaces because they have the following properties.

- COM interfaces are publicly defined. This means that any filter that implements the correct predefined interfaces will work in a filter graph without any knowledge about the other filters, because all filters are built with the same interface specifications.
- COM interfaces do not change after definition. A base set of interfaces are guaranteed to work; additional interfaces can be introduced to cover additional services. This definition prevents version problems.
- COM interfaces must have all methods implemented by any object that exposes them (even if the implemented method simply returns E_NOTIMPL). This assures that calling a method on the interface of an object will not generate an error.
- COM interfaces are discoverable. All COM objects support a method called QueryInterface that allows an external component to discover if an interface is present and retrieve a pointer to it.
- COM interfaces are implemented by the object that exposes the interface (they do not contain an implementation themselves). The interface is essentially a contract for the functionality. Objects like the filter graph manager, or Microsoft filters, have implemented interfaces that can be accessed. When you write a filter, you implement the interfaces.

To make filter development easier, DirectShow provides a set of C++ classes that help you implement the interfaces required by the objects you create.

File Formats

This section describes file formats in different files used in DirectShow, including the format of saved DirectShow graph files, DirectShow extensions to the AVI 2.0 file format, and how to register custom files so that the DirectShow file-reader filters can read them.

DirectShow Graph File Format

The format of a saved DirectShow™ graph file is as follows:

The docfile (storage file) contains a stream called ActiveMovieGraph. This single stream contains within it all the filters, filter names, file names, connections, and so on.

To load such a graph, either:

- Pass the storage file name to RenderFile. It will recognize that this is not a media file but a saved graph, and will restore the graph.

or:

1. Open the storage file (by using StgOpenStorage).
2. Query the filter graph manager for IPersistStream.
3. Open the L"ActiveMovieGraph" stream (by using IStorage::OpenStream).
4. Pass the stream to the filter graph (by using IPersistStream::Load).

The syntax of the graph within the stream follows:

```
<graph> ::= <version3><filters><connections><clock>END | <version2><filters><connec
<version3> ::= 0003\r\n
<version2> ::= 0002\r\n
<clock> ::= CLOCK <b> <required><b><clockid>\r\n
<required> ::= 1|0
<clockid> ::= <n>|<class id>
<filters ::= FILTERS <b>[<filter list><b>]
<connections> ::= CONNECTIONS [<b> <connection list>]
<filter list> ::= [<filter> <b>] <filter list>
<connection list> ::= [<connection><b>]<connection list>
<filter> ::= <n><b>"<name>"<b><class id><b>[<file>]<length><b1><filter data>
<file> ::= SOURCE "<name>"<b> | SINK "<name>"<b>
<class id> ::= class id of the filter in standard string form
<name> ::= any sequence of characters NOT including "
<length> ::= character string representing unsigned decimal number, for example, 23
             this is the number of bytes of data that follow the following space.
<b> ::= any combination of space, \t, \r, or \n
<b1> ::= exactly one space character
<n> ::= an identifier that will in fact be an integer, 0001, 0002, 0003, etc.
<connection> ::= <n1><b>"<pin1 id>"<b><n2><b>"<pin2 id>" <media type>
<n1> ::= identifier of first filter
<n2> ::= identifier of second filter
<pin1 id> ::= <name>
<pin2 id> ::= <name>
<media type> ::= <major type><b><sub type><b><flags><length><b1><format>
<major type> ::= <class id>
<sub type> ::= <class id>
<flags> ::= <FixedSizeSamples><b><TemporalCompression><b>
<FixedSizeSamples> ::= 1|0
<TemporalCompression> ::= 1|0
<Format> ::= <SampleSize><b><FormatType><b><FormatLength><b1><FormatData>
<FormatType> ::= class id of the format in standard string form
<FormatLength> ::= character string representing unsigned decimal number
             this is the number of bytes of data that follow the following space.
<FormatData> ::= binary data
```

On output there will be a new line (\r\n) per filter, one per connection, and one for each of the keywords FILTERS and CONNECTIONS. Each other case of <B> will be a single space. The keywords FILTERS, CONNECTIONS, and END are not localizable. Note also that the filter data and the format data are binary, so they might contain incorrect line breaks, null values, and so on.

The following approximates what the output looks like (a connection line is long and so has been split for presentation here, <with comments enclosed like this>).

```
0002
<version 2 of the syntax>
FILTERS
0001 "Source" {00000000-0000-0000-0000-000000000001} SOURCE "MyFile.mpg" 0000000000
<id   name   guid of the filter (need this to load it)   source file name   no priv
0002 "another filter" {00000000-0000-0000-0000-000000000002} 0000000008 XXXXXXXX
<id   name   guid   (this one is not a file source or sink)   8 bytes private data>
CONNECTIONS
0001 "Output pin" 0002 "In"   <no line break here>
<filter id pin id   filter id pin id  (output pin is first, then input pin)>
    0000000172 {00000000-0000-0000-0000-000000000003}   <no line break here>
```

```
<sample size,   media type major-type>
    {00000000-0000-0000-0000-000000000004} 1 0   <no line break here>
<media type sub-type, fixed size samples, no temporal compression>
    0000000093 {00000000-0000-0000-0000-000000000005} 18 YYYYYYYYYYYYYYYYYY
<length of format   format type   18 bytes of binary format data>
END
```

where:

- XXX... represents filter data
- YYY... represents format data

The strings and characters in the file are always in Unicode.

DV Data in the AVI File Format

Microsoft has specified the format for storage of digital video (DV) data in AVI files. Conforming to this specification will ensure that the AVI files authored in this format will be compatible with future versions of the Microsoft® DirectShow™ digital video architecture for the Microsoft Windows® platform.

This article provides background information to understand the format of audio-video interleaved (AVI) files containing DV audio and video data, or information for programmers who use DV-AVI files on other platforms. Applications that read or write AVI files should use the File Source (Async) filter with the AVI Splitter filter and the AVI MUX filter with the File Writer filter and their associated interfaces provided in the DirectShow architecture, rather than developing the routines to perform these services. These filters simplify the programming requirements for accessing these files.

This article also describes the format of AVI files containing DV data. Specific **FOURCC**s (four-character codes) for interleaved DV data streams and DV compressor/decompressor stream handlers are defined. The stream format structure for DV data is defined. Specifications for two methods of storing DV data in the AVI file format are specified.

It is assumed that the reader is familiar with the DV data format. (This format is defined in the *Specification of Consumer-use Digital VCRs*, also called the Blue Book).

Contents of this article:

For more information about resource interchange file format (RIFF) files, see the Windows Software Development Kit (SDK) *Multimedia Programmer's Guide* and *Multimedia Programmer's Reference*.

For more information about AVI files, see Chapter 6 of the Microsoft Video for Windows Development Kit version 1.1 *Programmer's Guide* and version 1.02 of the *OpenDML AVI File Format Extensions* published by the OpenDML AVI M-JPEG File Format Subcommittee, February 28, 1996.

For more information on compressors and decompressors, see the *Video Compression and Decompression Drivers* section of the Windows DDK Documentation in the MSDN Library.

Types of DV AVI Files

There are two types of DV AVI files:

- AVI Files Containing One DV Data Stream
- AVI Files Containing DV Video as a 'vids' Stream and DV Audio as 'auds' Streams

AVI Files Containing One DV Data Stream

Interleaved DV data can be stored in its native format as a single stream within an AVI RIFF file. This has the advantage of using the minimum amount of data storage for DV. The primary disadvantage is that this file format is not backward-compatible with Video for Windows, because it doesn't contain either a video 'vids' or an audio 'auds' stream. Support is provided for the interleaved DV stream through the DV Muxer and DV Splitter filters provided with DirectShow.

DV data can be stored in a single stream within an AVI RIFF file by specifying the 'iavs' (interleaved audio and video stream) **FOURCC** (four-character code) in the **fccType** member and either of the 'dvsd', 'dvhd', or 'dvsl' **FOURCC**s in the **fccHandler** member of the 'strh' stream header chunk. The frames per second of the video stream must be specified in the **dwRate** and **dwScale** members and the total number of video blocks in the 'movi' chunk in the **dwLength** member.

The 'dvsd' stream handler **FOURCC** specifies that the DV data is as defined in Part 2 of the *Specification of Consumer-use Digital VCRs*. Video is in the format of 525 lines at 29.97 Hz (525-60) or 625 lines at 25.00 Hz (625-50).

The 'dvhd' stream handler **FOURCC** specifies that the DV data is as defined in Part 3 of the *Specification of Consumer-use Digital VCRs*. Video is in the format of 1125 lines at 30.00 Hz (1125-60) or 1250 lines at 25.00 Hz (1250-50).

The 'dvsl' stream handler **FOURCC** specifies that the DV data is as defined in Part 6 of *Specification of Consumer-use Digital VCRs*. Video is in the format of high-compression SD (SDL).

Note The remainder of this article provides definitions for 'dvsd' streams.

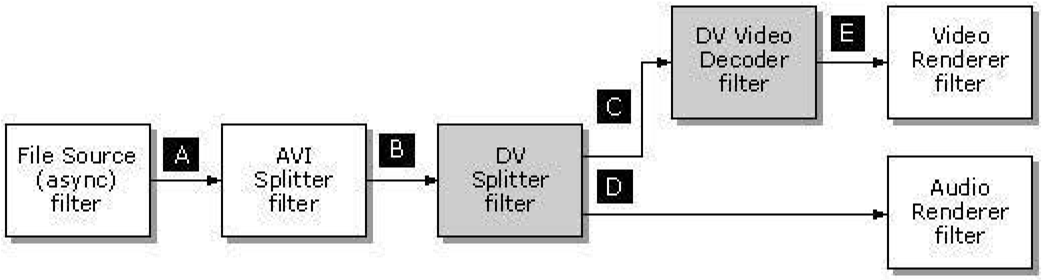
The stream header chunk must be followed by **DVINFO** stream format chunk. The **DVINFO** stream format has the following data structure defined for it:

```
typedef struct tag_DVINFO {
        DWORD dwDVAAuxSrc;
        DWORD dwDVAAuxCtl;
        DWORD dwDVAAuxSrc1;
        DWORD dwDVAAuxCtl1;
        DWORD dwDVVAuxSrc;
        DWORD dwDVVAuxCtl;
        DWORD dwDVReserved[2];
} DVINFO, *PDVINFO;
```

dwDVAAuxSrc	Specifies the Audio Auxiliary Data Source Pack for the first audio block (first 5 DV DIF sequences for 525-60 systems or 6 DV DIF sequences for 625-50 systems) of a frame. A DIF sequence is a data block that contains 150 DIF blocks. A DIF block consists of 80 bytes. The Audio Auxiliary Data Source Pack is defined in section D.7.1 of Part 2, Annex D, "The Pack Header Table and Contents of Packs" of the *Specification of Consumer-use Digital VCRs*.
dwDVAAuxCtl	Specifies the Audio Auxiliary Data Source Control Pack for the first audio block of a frame. The Audio Auxiliary Data Control Pack is defined in section D.7.2 of Part 2, Annex D, "The Pack Header Table and Contents of Packs" of the *Specification of Consumer-use Digital VCRs*.
dwDVAAuxSrc1	Specifies the Audio Auxiliary Data Source Pack for the second audio block (second 5 DV DIF sequences for 525-60 systems or 6 DV DIF sequences for 625-50 systems) of a frame.
dwDVAAuxCtl1	Specifies the Audio Auxiliary Data Source Control Pack for the second audio block of a frame.
dwDVVAuxSrc	Specifies the Video Auxiliary Data Source Pack as defined in section D.8.1 of Part 2, Annex D, "The Pack Header Table and Contents of Packs" of the *Specification of Consumer-use Digital VCRs*.
dwDVVAuxCtl	Specifies the Video Auxiliary Data Source Control Pack as defined in section D.8.2 of Part 2, Annex D, "The Pack Header Table and Contents of Packs" of the *Specification of Consumer-use Digital VCRs*.
DwDVReserved [2]	Reserved. Set this array to zero.

The actual DV data is stored as '##dc' chunks in the 'movi' chunk (the ## in the format represents the stream identifier). Each chunk contains one frame of data, either 10 or 12 DV DIF sequences for 525-60 or 625-50 systems, respectively. The DV SD ('dvsd') DIF sequence format is defined in Part 2 of the *Specification of Consumer-use Digital VCRs*.

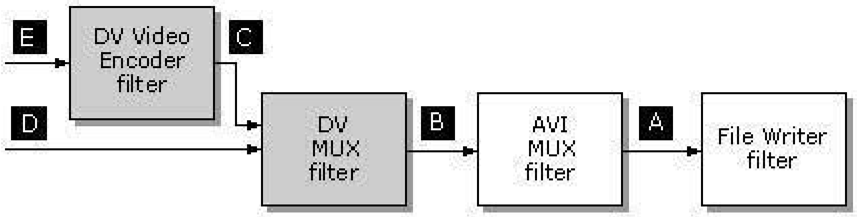
The following diagram illustrates the playback of an AVI file with one DV data stream using a DirectShow filter graph (the DV Splitter and DV Video Decoder filters are included in DirectShow specifically to deal with DV data). The table that follows the diagram defines the media types.



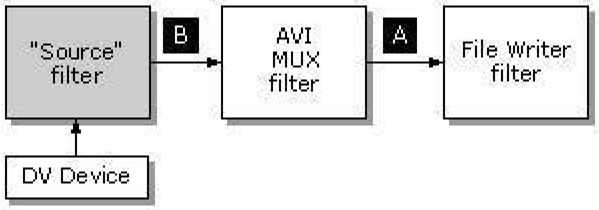
DV media types table

Media	Major type	Subtype	Format structure
A	MEDIATYPE_Stream	MEDIASUBTYPE_AVI	none
B	MEDIATYPE_iavs	MEDIASUBTYPE_dvsd, MEDIASUBTYPE_dvhd, or MEDIASUBTYPE_dvsl	DVINFO
C	MEDIATYPE_VIDEO	MEDIASUBTYPE_dvsd, MEDIASUBTYPE_dvhd, or MEDIASUBTYPE_dvsl	DVINFO
D	MEDIATYPE_AUDIO	NULL	WAVEFORMATEX
E	MEDIATYPE_VIDEO	standard video types	VIDEOINFO

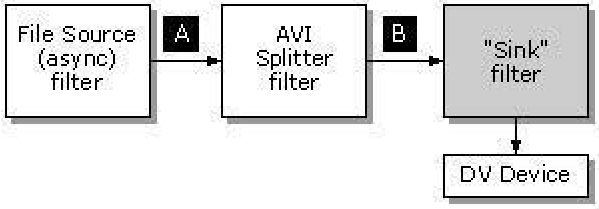
The following diagram illustrates the creation of an AVI file with one DV data stream by using a DirectShow filter graph (the DV Video Encoder and DV Muxer filters are included in DirectShow specifically to deal with DV data). The preceding table defines the media types. Upstream filters (not shown) can be of any combination to produce the proper media types, D and E.



The following diagram illustrates the creation of an AVI file with one DV data stream using a source filter that communicates through hardware device drivers with a DV device (such as a 1394-based DV camcorder) for DV data input, by using a DirectShow filter graph (the source filter is included in DirectShow specifically to deal with DV data). The preceding table defines the media types.



The following diagram illustrates the output of an AVI file with one DV data stream using a sink filter that communicates through hardware device drivers with a DV device (such as a 1394-based DV camcorder) for DV data output, by using a DirectShow filter graph (the sink filter is included in DirectShow specifically to deal with DV data). The preceding table defines the media types.



The following example shows the AIFF RIFF form for an AVI file with one DV data stream, expanded with completed header chunks:

```
00000000 RIFF (0FAE35D4) 'AVI '
0000000C     LIST (00000106) 'hdrl'
00000018         avih (00000038)
                     dwMicroSecPerFrame    : 33367
                     dwMaxBytesPerSec      : 3728000
                     dwPaddingGranularity  : 0
                     dwFlags               : 0x810 HASINDEX | TRUSTCKTYPE
                     dwTotalFrames         : 2192
                     dwInitialFrames       : 0
                     dwStreams             : 1
                     dwSuggestedBufferSize : 120000
                     dwWidth               : 720
                     dwHeight              : 480
                     dwReserved            : 0x0
00000058         LIST (0000006C) 'strl'
00000064             strh (00000038)
                         fccType                : 'iavs'
                         fccHandler             : 'dvsd'
                         dwFlags                : 0x0
                         wPriority              : 0
                         wLanguage              : 0x0 undefined
                         dwInitialFrames        : 0
                         dwScale                : 100 (29.970 Frames/Sec)
                         dwRate                 : 2997
                         dwStart                : 0
                         dwLength               : 2192
                         dwSuggestedBufferSize  : 120000
                         dwQuality              : 0
                         dwSampleSize           : 0
                         rcFrame                : 0,0,720,480
000000A4             strf (00000020)
                         dwDVAAuxSrc      : 0x........
                         dwDVAAuxCtl      : 0x........
                         dwDVAAuxSrc1     : 0x........
                         dwDVAAuxCtl1     : 0x........
                         dwDVVAuxSrc      : 0x........
                         dwDVVAuxCtl      : 0x........
                         dwDVReserved[2]  : 0,0
000000CC     LIST (0FADAC00) 'movi'
0FADACD4     idx1 (00008900)
```

AVI Files Containing DV Video as a 'vids' Stream and DV Audio as 'auds' Streams

Interleaved DV data can be split into a video stream and one to four audio streams within an AVI RIFF file. This has the advantage of being backward-compatible with Video for Windows, because it contains a standard video 'vids' stream and at least one standard audio 'auds' stream The primary disadvantage is that this file format requires the audio data to be redundantly stored as audio streams. The "video" stream is actually the native interleaved DV data stream. However, as a standard 'vids' stream with a handler type of 'dvsd', the DV Video Decoder is used. This format also requires that "captured" files are split by using the DV Splitter filter before they are written as AVI files.

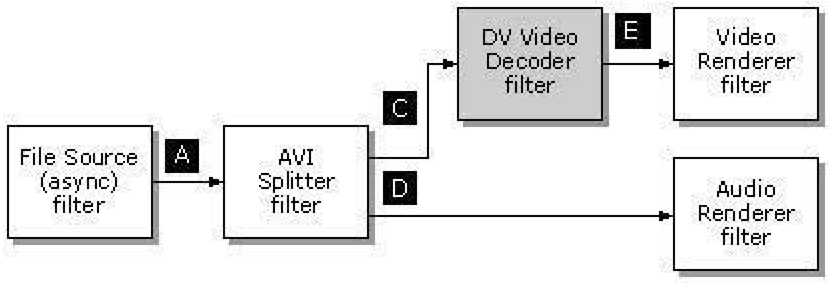
DV data can be stored as a video stream with a separate number of audio streams in an AVI RIFF file. The video stream is specified with a standard video stream header (the **fccType** member value is 'vids'). The **fccHandler** member is specified as 'dvsd', 'dvhd', or 'dvsl'. The frames per second of the video stream must be specified in the **dwRate** and **dwScale** members and the total number of video blocks in the 'movi' chunk in the **dwLength** member.

In this AVI file containing DV video as a 'vids' stream and DV audio as 'auds' streams form of DV, the video stream format chunk is a standard BITMAPINFOHEADER structure. The stream format chunk can be optionally extended to include the DVINFO structure, by increasing the stream format chunk size from 40 bytes (size of the **BITMAPINFOHEADER** structure) to 72 bytes (size of **BITMAPINFOHEADER** plus **DVINFO** structures) and immediately following the **BITMAPINFOHEADER** data structure with a **DVINFO** data structure.

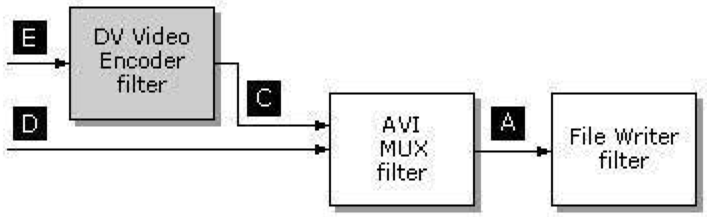
The audio stream(s) is specified with a standard audio stream header (the **fccType** member value is 'auds'). The **fccHandler** member is not used for audio streams.

The DV video data is stored as '##dc' chunks, as defined in the preceding description of an AVI file with one DV data, and the audio data is stored as '##wb' chunks in the 'movi' chunk.

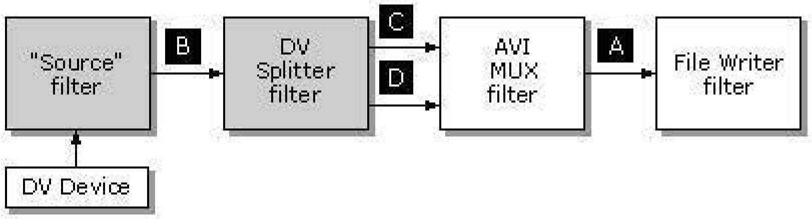
The following diagram illustrates the playback of an AVI file containing DV video as a 'vids' stream and DV audio as 'auds' streams, by using a DirectShow filter graph (the DV Video Decoder filter is included in DirectShow specifically to deal with DV data). The DV media types table defines the media types.



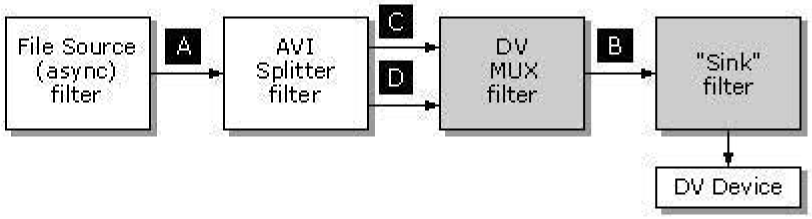
The following diagram illustrates the creation of an AVI file containing DV video as a 'vids' stream and DV audio as 'auds' streams, using a DirectShow filter graph (the DV Video Decoder is included in DirectShow specifically to deal with DV data). The DV media types table defines the media types. Upstream filters (not shown) can be of any combination to produce the proper media types, D and E.



The following diagram illustrates the creation of an AVI file containing DV video as a 'vids' stream and DV audio as 'auds' streams using a source filter that communicates through hardware device drivers with a DV device (such as a 1394-based DV camcorder) for DV data input, by using a DirectShow filter graph (the source and DV Splitter filters are included in DirectShow specifically to deal with DV data). The DV media types table defines the media types.



The following diagram illustrates the output of an AVI file containing DV video as a 'vids' stream and DV audio as 'auds' streams using a sink filter that communicates through hardware device drivers with a DV device (such as a 1394-based DV camcorder) for DV data output, by using a DirectShow filter graph (the DV Muxer and sink filters are included in DirectShow specifically to deal with DV data). The DV media types table defines the media types.



The following example shows the AIFF RIFF form for an AVI file containing DV video as a 'vids' stream and DV audio as 'auds' streams expanded with completed header chunks (including optional DVINFO data following the BITMAPINFO in the 'strf' sub-chunk for the 'vids' stream):

```
00000000 RIFF (103E2920) 'AVI '
0000000C     LIST (00000146) 'hdrl'
00000018         avih (00000038)
                     dwMicroSecPerFrame    : 33367
                     dwMaxBytesPerSec      : 3728000
                     dwPaddingGranularity  : 0
                     dwFlags               : 0x810 HASINDEX | TRUSTCKTYPE
                     dwTotalFrames         : 2192
                     dwInitialFrames       : 0
                     dwStreams             : 2
                     dwSuggestedBufferSize : 120000
                     dwWidth               : 720
                     dwHeight              : 480
                     dwReserved            : 0x0
00000058         LIST (00000094) 'strl'
00000064             strh (00000038)
                         fccType               : 'vids'
                         fccHandler            : 'dvsd'
                         dwFlags               : 0x0
                         wPriority             : 0
                         wLanguage             : 0x0 undefined
                         dwInitialFrames       : 0
                         dwScale               : 100 (29.970 Frames/Sec)
                         dwRate                : 2997
                         dwStart               : 0
                         dwLength              : 2192
                         dwSuggestedBufferSize : 120000
                         dwQuality             : 0
                         dwSampleSize          : 0
                         rcFrame               : 0,0,720,480
000000A4             strf (00000048)
                         biSize          : 40
```

```
                           biWidth         : 720
                           biHeight        : 480
                           biPlanes        : 1
                           biBitCount      : 24
                           biCompression   : 0x64737664 'dvsd'
                           biSizeImage     : 120000
                           biXPelsPerMeter : 0
                           biYPelsPerMeter : 0
                           biClrUsed       : 0
                           biClrImportant  : 0
                           dwDVAAuxSrc     : 0x........
                           dwDVAAuxCtl     : 0x........
                           dwDVAAuxSrc1    : 0x........
                           dwDVAAuxCtl1    : 0x........
                           dwDVVAuxSrc     : 0x........
                           dwDVVAuxCtl     : 0x........
                           dwDVReserved[2] : 0,0
000000F4           LIST (0000005E) 'strl'
00000100               strh (00000038)
                           fccType                : 'auds'
                           fccHandler             : '    '
                           dwFlags                : 0x0
                           wPriority              : 0
                           wLanguage              : 0x0 undefined
                           dwInitialFrames        : 0
                           dwScale                : 1 (32000.000 Samples/Sec)
                           dwRate                 : 32000
                           dwStart                : 0
                           dwLength               : 2340474
                           dwSuggestedBufferSize  : 4272
                           dwQuality              : 0
                           dwSampleSize           : 4
                           rcFrame                : 0,0,0,0
00000140               strf (00000012)
                           wFormatTag      : 1 PCM
                           nChannels       : 2
                           nSamplesPerSec  : 32000
                           nAvgBytesPerSec : 128000
                           nBlockAlign     : 4
                           wBitsPerSample  : 16
                           cbSize          : 0
00000814       LIST (103D0EF4) 'movi'
103D1710       idx1 (00011210)
```

AVI RIFF File Reference

The Microsoft audio-video interleaved (AVI) file format is a RIFF file specification used with applications that capture, edit, and play back audio-video sequences. In general, AVI files contain multiple streams of different types of data. Most AVI sequences use both audio and video streams. A simple variation for an AVI sequence uses video data and does not require an audio stream.

Modifications to the original AVI file specification made in the OpenDML AVI File Format Extensions are not discussed in this section. For further information on these extensions, see version 1.02 of the *OpenDML AVI File Format Extensions* published by the OpenDML AVI M-JPEG File Format Subcommittee, February 28, 1996.

This section contains the following topics:

- AVI RIFF Form

- AVI Main Header
- AVI Stream Headers
- Stream Data (LIST 'movi' Chunk)
- BITMAPINFOHEADER Structure
- WAVEFORMATEX Structure

AVI RIFF Form

AVI files use the AVI RIFF form. The AVI RIFF form is identified by the **FOURCC** (four-character code) 'AVI '. All AVI files include two mandatory LIST chunks. These chunks define the format of the stream and stream data. AVI files might also include an index chunk. This optional chunk specifies the location of data chunks within the file. An AVI file with these components has the following form:

```
RIFF ('AVI '
      LIST ('hdrl'
            .
            .
            .
           )
      LIST ('movi'
            .
            .
            .
           )
      ['idx1'<AVI Index>]
     )
```

The LIST chunks and the index chunk are subchunks of the RIFF 'AVI ' chunk. The 'AVI ' chunk identifies the file as an AVI RIFF file. The LIST 'hdrl' chunk defines the format of the data and is the first required LIST chunk. The LIST 'movi' chunk contains the data for the AVI sequence and is the second required LIST chunk. The 'idx1' chunk is the index chunk. AVI files must keep these three components in the proper sequence.

The LIST 'hdrl' and LIST 'movi' chunks use subchunks for their data. The following example shows the AVI RIFF form expanded with the chunks needed to complete the LIST 'hdrl' and LIST 'movi' chunks:

```
RIFF ('AVI '
      LIST ('hdrl'
            'avih'(<Main AVI Header>)
            LIST ('strl'
                  'strh'(<Stream header>)
                  'strf'(<Stream format>)
                  'strd'(<additional header data>)
                  'strn'(<Stream name>)
                  ...
                 )
             .
             .
             .
           )

      LIST ('movi'
            {SubChunk | LIST ('rec '
                              SubChunk1
                              SubChunk2
```

```
                                        .
                                        .
                                        .
                                    )
                        .
                        .
                        .
                    }
                    .
                    .
                    .
                  )
              ['idx1'<AVI Index>]
            )
```

AVI Main Header

This and following sections describe the chunks contained in the LIST 'hdrl' and LIST 'movi' chunks. The 'idx1' chunk is not described in this document. For more information on the 'idx1' chunk and indexes in AVI files, see version 1.02 of the *OpenDML AVI File Format Extensions* published by the OpenDML AVI M-JPEG File Format Subcommittee, February 28, 1996.

The file begins with the main header. In the AVI file, this header is identified by the 'avih' **FOURCC** (four-character code). The header contains global information for the entire AVI file, such as the number of streams within the file and the width and height of the AVI sequence. The AVI main header structure is defined as follows:

```
typedef struct {
        DWORD dwMicroSecPerFrame;
        DWORD dwMaxBytesPerSec;
        DWORD dwReserved1;
        DWORD dwFlags;
        DWORD dwTotalFrames;
        DWORD dwInitialFrames;
        DWORD dwStreams;
        DWORD dwSuggestedBufferSize;
        DWORD dwWidth;
        DWORD dwHeight;
        DWORD dwReserved[4];
} MainAVIHeader;
```

dwMicroSecPerFrame	Specifies the number of microseconds between frames. This value indicates the overall timing for the file.
dwMaxBytesPerSec	Specifies the approximate maximum data rate of the file. This value indicates the number of bytes per second the system must handle to present an AVI sequence as specified by the other parameters contained in the main header and stream header chunks.
dwReserved1	Reserved. Set this to zero.
dwFlags	Contains any flags for the file. The following flags are defined: AVIF_HASINDEX — Indicates the AVI file has an 'idx1' chunk containing an index at the end of the file. For good performance, all AVI files should contain an index.

	AVIF_MUSTUSEINDEX — Indicates that the index, rather than the physical ordering of the chunks in the file, should be used to determine the order of presentation of the data. For example, you could use this to create a list of frames for editing.
	AVIF_ISINTERLEAVED — Indicates the AVI file is interleaved.
	AVIF_WASCAPTUREFILE — Indicates the AVI file is a specially allocated file used for capturing real-time video. Applications should warn the user before writing over a file with this flag set because the user probably defragmented this file.
	AVIF_COPYRIGHTED — Indicates the AVI file contains copyrighted data and software. When this flag is used, software should not permit the data to be duplicated.
dwTotalFrames	Specifies the total number of frames of data in the file.
dwInitialFrames	Specifies the initial frame for interleaved files. Noninterleaved files should specify zero. If you are creating interleaved files, specify the number of frames in the file prior to the initial frame of the AVI sequence in this member. For more information about the contents of this member, see "Special Information for Interleaved Files" in the Video for Windows Programmer's Guide.
dwStreams	Specifies the number of streams in the file. For example, a file with audio and video has two streams.
dwSuggestedBufferSize	Specifies the suggested buffer size for reading the file. Generally, this size should be large enough to contain the largest chunk in the file. If set to zero, or if it is too small, the playback software will have to reallocate memory during playback, which will reduce performance. For an interleaved file, the buffer size should be large enough to read an entire record, and not just a chunk.
dwWidth	Specifies the width of the AVI file in pixels.
dwHeight	Specifies the height of the AVI file in pixels.
dwReserved[4]	Reserved. Set this array to zero.

AVI Stream Headers

The main header is followed by one or more 'strl' chunks. (A 'strl' chunk is required for each data stream.) These chunks contain information about the streams in the file. Each 'strl' chunk must contain a stream header and stream format chunk. Stream header chunks are identified by the **FOURCC** (four-character code) 'strh' and the stream format chunks are identified by the **FOURCC** 'strf'. In addition to the stream header and stream format chunks, the 'strl' chunk might also contain a stream-header data chunk and a stream name chunk. Stream-header data chunks are identified by the **FOURCC** 'strd'. Stream name chunks are identified by the **FOURCC** 'strn'.

The stream header structure contains header information for a single stream of a file.

```
typedef struct {
        FOURCC fccType;
        FOURCC fccHandler;
        DWORD  dwFlags;
        DWORD  dwPriority;
        DWORD  dwInitialFrames;
        DWORD  dwScale;
        DWORD  dwRate;
        DWORD  dwStart;
        DWORD  dwLength;
        DWORD  dwSuggestedBufferSize;
        DWORD  dwQuality;
        DWORD  dwSampleSize;
        RECT   rcFrame;
} AVIStreamHeader;
```

The stream header specifies the type of data the stream contains, such as audio or video, by means of a **FOURCC**.

fccType	Contains a **FOURCC** that specifies the type of the data contained in the stream. The following standard AVI values for video and audio are defined: 'vids' — Indicates the stream contains video data. The stream format chunk contains a BITMAPINFO structure that can include palette information. 'auds' — Indicates the stream contains audio data. The stream format chunk contains a WAVEFORMATEX or PCMWAVEFORMAT structure. 'txts' — Indicates the stream contains text data.
fccHandler	Optionally, contains a **FOURCC** that identifies a specific data handler. The data handler is the preferred handler for the stream. For audio and video streams, this specifies the installable compressor or decompressor.
dwFlags	Contains any flags for the data stream. The bits in the high-order word of these flags are specific to the type of data contained in the stream. The following standard flags are defined: AVISF_DISABLED — Indicates this stream should not be enabled by default. AVISF_VIDEO_PALCHANGES — Indicates this video stream contains palette changes. This flag warns the playback software that it will need to animate the palette.
dwPriority	Specifies priority of a stream type. For example, in a file with multiple audio streams, the one with the highest priority might be the default stream.
dwInitialFrames	Specifies how far audio data is skewed ahead of the video frames in interleaved files. Typically, this is about 0.75 seconds. If you are creating interleaved files, specify the number of frames in the file prior to the initial frame of the AVI sequence in this member. For more information about the contents of this member, see "Special Information for Interleaved Files" in the Video for Windows Programmer's Guide.
dwScale	Used with **dwRate** to specify the time scale that this stream will use. Dividing **dwRate** by **dwScale** gives the number of samples per second. For video streams, this rate should be the frame rate. For audio streams, this rate should correspond to the time needed for **nBlockAlign** bytes of audio, which for PCM audio simply reduces to the sample rate.
dwRate	See **dwScale**.
dwStart	Specifies the starting time of the AVI file. The units are defined by the **dwRate** and **dwScale** members in the main file header. Usually, this is zero, but it can specify a delay time for a stream that does not start concurrently with the file.

dwLength	Specifies the length of this stream. The units are defined by the **dwRate** and **dwScale** members of the stream's header.
dwSuggestedBufferSize	Specifies how large a buffer should be used to read this stream. Typically, this contains a value corresponding to the largest chunk present in the stream. Using the correct buffer size makes playback more efficient. Use zero if you do not know the correct buffer size.
dwQuality	Specifies an indicator of the quality of the data in the stream. Quality is represented as a number between 0 and 10,000. For compressed data, this typically represents the value of the quality parameter passed to the compression software. If set to –1, drivers use the default quality value.
dwSampleSize	Specifies the size of a single sample of data. This is set to zero if the samples can vary in size. If this number is nonzero, then multiple samples of data can be grouped into a single chunk within the file. If it is zero, each sample of data (such as a video frame) must be in a separate chunk. For video streams, this number is typically zero, although it can be nonzero if all video frames are the same size. For audio streams, this number should be the same as the **nBlockAlign** member of the WAVEFORMATEX structure describing the audio.
rcFrame	Specifies the destination rectangle for a text or video stream within the movie rectangle specified by the **dwWidth** and **dwHeight** members of the AVI main header structure. The **rcFrame** member is typically used in support of multiple video streams. Set this rectangle to the coordinates corresponding to the movie rectangle to update the whole movie rectangle. Units for this member are pixels. The upper-left corner of the destination rectangle is relative to the upper-left corner of the movie rectangle.

The last eight members describe the playback characteristics of the stream. These factors include the playback rate (**dwScale** and **dwRate**), the starting time of the sequence (**dwStart**), the length of the sequence (**dwLength**), the size of the playback buffer (**dwSuggestedBuffer**), an indicator of the data quality (**dwQuality**), and the sample size (**dwSampleSize**).

Some of the members in the stream header structure are also present in the main header structure. The data in the main header applies to the whole file, while the data in the stream header structure applies only to a stream.

A stream format ('strf') chunk must follow a stream header ('strh') chunk. The stream format chunk describes the format of the data in the stream. For video streams, the information in this chunk is a BITMAPINFO structure (including palette information if appropriate). For audio streams, the information in this chunk is a WAVEFORMATEX or PCMWAVEFORMAT structure. (The **WAVEFORMATEX** structure is an extended version of the WAVEFORMAT structure.) For more information about this structure and other stream types, see the *New Multimedia Data Types and Data Techniques Standards Update*.

The 'strl' chunk might also contain an additional stream-header data ('strd') chunk. If used, this chunk follows the stream format chunk. The format and content of this chunk is defined by installable compression or decompression drivers. Typically, drivers use this information for configuration. Applications that read and write RIFF files do not need to decode this information. They transfer this data to and from a driver as a memory block.

The optional 'strn' stream name chunk provides a zero-terminated text string describing the stream. (The AVI file functions can use this chunk to let applications identify the streams they want to access by their names.)

An AVI player associates the stream headers in the LIST 'hdrl' chunk with the stream data in the LIST 'movi' chunk by using the order of the 'strl' chunks. The first 'strl' chunk applies to stream 0, the second applies to stream 1, and so forth.

For example, if the first 'strl' chunk describes the wave audio data, the wave audio data is contained in stream 0. Similarly, if the second 'strl' chunk describes video data, then the video data is contained in stream 1.

Stream Data (LIST 'movi' Chunk)

Following the header information is a LIST 'movi' chunk that contains chunks of the actual data in the streams — that is, the pictures and sounds themselves. The data chunks can reside directly in the LIST 'movi' chunk or they might be grouped into 'rec' chunks. The 'rec' grouping implies that the grouped chunks should be read from disk all at once. This is used only for files specifically interleaved to play from CD-ROM.

Like any RIFF chunk, the data chunks contain a **FOURCC** (four-character code) to identify the chunk type. A **FOURCC** is a 32-bit quantity represented as a sequence of one to four ASCII alphanumeric characters, padded on the right with blank characters. The **FOURCC** that identifies each chunk consists of the stream number and a two-character code that defines the type of information encapsulated in the chunk. For example, a waveform chunk is identified by a two-character code of 'wb'. If a waveform chunk corresponded to the second LIST 'hdrl' stream description, it would have a **FOURCC** of '01wb'.

Note While two-character codes are a convenient way to describe a stream, do not expect them to be recognized by other applications. Use **FOURCC**s when creating a stream or transferring the information to other applications.

Because all the format information is in the header, the audio data contained in these data chunks does not contain any information about its format. An audio data chunk has the following format (the ## in the format represents the stream identifier):

```
WAVE Bytes '##wb'
     BYTE  abBytes[];
```

Video data can be compressed or uncompressed DIBs. An uncompressed DIB has BI_RGB specified for the **biCompression** member in its associated BITMAPINFO structure. A compressed DIB has a value other than BI_RGB specified in the **biCompression** member. For more information about compression formats, see the description of the BITMAPINFOHEADER data structure in the Microsoft Windows Programmer's Reference.

A data chunk for an uncompressed DIB contains RGB video data. These chunks are identified by a two-character code of 'db' (db is an abbreviation for DIB bits). Data chunks for a compressed DIB are identified by a two-character code of 'dc' (dc is an abbreviation for DIB compressed). Neither data chunk will contain any header information about the DIBs. The data chunk for an uncompressed DIB has the following form:

```
DIB  Bits  '##db'
     BYTE  abBits[];
```

The data chunk for a compressed DIB has the following form.

```
Compressed DIB  Bits  '##dc'
                BYTE  abBits[];
```

Video data chunks can also define new palette entries used to update the palette during an AVI sequence. For more information on specifying palette information, see *Video for Windows Programmer's Guide*.

Text streams can use arbitrary two-character codes.

BITMAPINFOHEADER Structure

The **BITMAPINFOHEADER** structure contains information for the video stream of an AVI RIFF file. This structure has the following members:

```
typedef struct tagBITMAPINFOHEADER {
        DWORD  biSize;
        LONG   biWidth;
        LONG   biHeight;
        WORD   biPlanes;
        WORD   biBitCount;
        DWORD  biCompression;
        DWORD  biSizeImage;
        LONG   biXPelsPerMeter;
        LONG   biYPelsPerMeter;
        DWORD  biClrUsed;
        DWORD  biClrImportant;
} BITMAPINFOHEADER;
```

biSize	Specifies the number of bytes required by the structure.
biWidth	Specifies the width of the bitmap, in pixels.
biHeight	Specifies the height of the bitmap, in pixels. If **biHeight** is positive, the bitmap is a bottom-up DIB (device-independent bitmap) and its origin is the lower left corner. If **biHeight** is negative, the bitmap is a top-down DIB and its origin is the upper left corner.
biPlanes	Specifies the number of planes for the target device. This value must be set to 1.
biBitCount	Specifies the number of bits per pixel. Some compression formats need this information to properly decode the colors in the pixel.
biCompression	Specifies the type of compression used or requested. Both existing and new compression formats use this member.
biSizeImage	Specifies the size, in bytes, of the image. This can be set to 0 for uncompressed RGB bitmaps.
biXPelsPerMeter	Specifies the horizontal resolution, in pixels per meter, of the target device for the bitmap. An application can use this value to select a bitmap from a resource group that best matches the characteristics of the current device.
biYPelsPerMeter	Specifies the vertical resolution, in pixels per meter, of the target device for the bitmap.

biClrUsed	Specifies the number of color indices in the color table that are actually used by the bitmap. If this value is zero, the bitmap uses the maximum number of colors corresponding to the value of the **biBitCount** member for the compression mode specified by **biCompression**.
biClrImportant	Specifies the number of color indices that are considered important for displaying the bitmap. If this value is zero, all colors are important.

When the value in the **biBitCount** member is set to greater than eight, video drivers can assume bitmaps are true color and they do not use a color table.

When the value in the **biBitCount** member is set to less than or equal to eight, video drivers can assume the bitmap uses a palette or color table defined in the **BITMAPINFO** data structure. This data structure has the following members:

```
typedef struct tagBITMAPINFO {
        BITMAPINFOHEADER bmiHeader;
        RGBQUAD          bmiColors[1];
} BITMAPINFO;
```

The BITMAPINFO **bmiheader** member specifies a **BITMAPINFOHEADER** structure. The BITMAPINFO **bmiColors** member specifies an array of **RGBQUAD** data types that define the colors in the bitmap.

WAVEFORMATEX Structure

The **WAVEFORMATEX** structure contains information for the audio stream(s) of an AVI RIFF file. This structure has the following members:

```
typedef struct waveformat_extended_tag {
        WORD  wFormatTag;
        WORD  nChannels;
        DWORD nSamplesPerSec;
        DWORD nAvgBytesPerSec;
        WORD  nBlockAlign;
        WORD  wBitsPerSample;
        WORD  cbSize;
} WAVEFORMATEX;
```

wFormatTag	Defines the audio waveform type of the audio stream. A complete list of format tags can be found in the MMREG.H header file included with Microsoft Visual C++ and other Microsoft products.
nChannels	Specifies the number of channels in the audio stream, 1 for mono, 2 for stereo.
nSamplesPerSec	Specifies the frequency of the sample rate of the audio stream in samples/second (Hz). Examples are 11,025, 22,050, or 44,100.
nAvgBytesPerSec	Specifies the average data rate. Playback software can estimate the buffer size by using this value.
nBlockAlign	Specifies the block alignment of the data, in bytes. Playback software must process a multiple of **nBlockAlign** bytes of data at a time, so that the value of **nBlockAlign** can be used for buffer alignment.

wBitsPerSample	Specifies the number of bits per sample per channel data. Each channel is assumed to have the same sample resolution. If this field is not needed, then you should set it to zero.
cbSize	Specifies the size, in bytes, of the extra information in the format header, not including the size of the **WAVEFORMATEX** structure. For example, in the wave format corresponding to the **wFormatTag** WAVE_FORMAT_IMA_ADPCM, **cbSize** is calculated as sizeof (IMAADPCMWAVEFORMAT) - sizeof(WAVEFORMATEX), which yields two.

AVI 2.0 File Format Extensions

DirectShow currently supports the following AVI 2.0 file format extensions:

- Increased AVI file size (greater than 1 GB)
- Hierarchical indexing

See the specification in version 1.02 of the *OpenDML AVI File Format Extensions* published by the OpenDML AVI M-JPEG File Format Subcommittee, February 28, 1996.

Registering a Custom File Type

This topic describes how to register a new file type so that file-reader source filters can recognize it. The mechanism used here is taken from the Microsoft® Win32® GetClassFile function, which is used to return the CLSID associated with the given file name. Microsoft DirectShow™ media types use the same quadruple set of values in the registry that are used for **GetClassFile** FileType registrations, but associate a file matching this criteria with a media type rather than a file type. Also, the registry entry for a DirectShow media type provides the CLSID of a source filter that can be used to read this media type.

For both FileType and MediaType registration, a pattern in the registry contains a series of entries of the form:

```
regdb key = offset, cb, mask, value
```

The media type is defined as a CLSID pair, {Majortype clsid, Subtype clsid}. If the data in the file at the specified offset or offsets matches a pattern in HKEY_CLASSES_ROOT\Media Type\{<major type>}\{<subtype>}, the media type CLSID pair associated with that pattern is the media type of the file.

The parameters of the registry key are interpreted as follows. The value of the *offset* item is an offset from the beginning or end of the file, and the *cb* item is a length in bytes. These two decimal values represent a particular byte range in the file. (A negative value for the offset item is interpreted from the end of the file.) The *mask* value is a hexadecimal bit mask that is used to perform a logical AND operation, with the byte range specified by *offset* and *cb*. The result of the logical AND operation is compared with the *value* item. If the mask is omitted, it is assumed to be all ones. The number of hexadecimal digits in *mask* and *value* must be twice the value of *cb* (because *cb* is in bytes).

Each pattern in the registry is compared to the file in the order of the patterns in the database. The first pattern where each of the value items matches the result of the AND operation determines the media type of the file.

Note that each entry can have multiple quadruples, all of which must match the data in the file for the media type to be associated with the file. An example of using multiple quadruples in a single entry might be to match the byte sequence at the beginning and at the end of the file. The following example shows a pattern of AB CD 12 34 as the first 4 bytes in the file and AB AB 00 AB as the last 4 bytes in the file (no masks applied here). All elements must match for the pattern to match a file with a media type.

```
0 = REG_SZ 0, 4, , ABCD1234,  -4, 4, , ABAB00AB
```

Additionally, there can be multiple entries specified under a single media type, a match to any one of which will associate the file with the media type.

For example, the pattern contained in the following entries of the registry requires that the first three bytes be AB CD 12, that the fourth byte be 32, 33, 34, or 35, and that the last 4 bytes be FE FE FE FE:

```
HKEY_CLASSES_ROOT
    Media Type
         {12345678-0000-0001-C000-000000000095}
                {87654321-0000-0001-C000-000000000095}
                    0 = REG_SZ 0, 4, FFFFFFFE, ABCD1234, -4, 4, , FEFEFEFE
                    1 = REG_SZ 0, 4, FFFFFFFE, ABCD1232, -4, 4, , FEFEFEFE
                    Source Filter = {56781234-0000-0001-C000-000000000095}
```

If a file contains such a pattern, the media type {12345678-0000-0001-C000-000000000095} {87654321-0000-0001-C000-000000000095} will be associated with this file. The file source filter for the media type is identified by the CLSID of the Source Filter value under the key for the media type.

The media type can be used to find filter handlers for the file in order to render it. A handler for a type performs a more exact test of the file to be sure of the type before attempting to render the data.

Note that this scheme allows for a set of alternative masks (for instance, .wav files) that might or might not have a RIFF header.

Transform Filters

This section describes how to create a transform filter, types of transform filters, how to use the transform base classes, which base class member functions to override and when, and how to connect transform filters.

Creating a Transform Filter

Transform filters transform the media data that comes into their input pins and send the transformed data out their output pins. Transform filters can be used to compress and decompress data, to split audio and visual data, or to apply effects, such as contrast or warbling, to media data. DirectShow contains several sample transform filters that perform different kinds of transformations. See DirectShow Filters for a description of the transform filters supplied by DirectShow. See Write a Transform Filter in C/C++ for instructions on how to write your own transform filters in C++. See Using the CTransformFilter and CTransInPlaceFilter Transform Base Classes for a discussion of the CTransformFilter and CTransInPlaceFilter transform filter base classes. See Connecting Transform Filters for a discussion of connecting to a transform filter.

This article steps through the process of creating a transform filter for a Microsoft® DirectShow™ filter graph that uses the DirectShow C++ class library. It covers five basic steps, and in the last step shows how to override the required member functions in your derived class to implement the transform filter. It answers two common questions that arise when creating transform filters: Which base class do I use? and How do I override member functions?

Contents of this article:

Writing a Transform Filter

Writing a transform filter can be broken into the following discrete steps.

1. Determine if the filter must copy media samples or can handle them in place.

   The fewer copies in the media stream, the better. However, some filters require a copy operation; this influences the choice of base classes.

2. Determine which base classes to use and derive the filter class (and pin classes, if necessary) from the base classes.

   In this step, create the header or headers for your filter. In many cases, you can use the transform base classes, derive your class from the correct transform filter class, and override a few member functions. In other cases, you can use the more generic base classes. These classes implement most of the connection and negotiation mechanism; but these classes also allow more flexibility at the cost of overriding more member functions.

3. Add the code necessary to instantiate the filter.

   This step requires adding a static **CreateInstance** member function to your derived class and also a global array that contains the name of the filter, a CLSID, and a pointer to that member function.

4. Add a **NonDelegatingQueryInterface** member function to pass out any unique interfaces in your filter.

   This step addresses the Component Object Model (COM) aspects of implementing interfaces, other than those in the base classes.

5. Override the appropriate base class member functions.

   This includes writing the transform function that is unique to your filter and overriding a few member functions that are necessary for the connection process, such as setting the allocator size or providing media types.

Determine if the Filter Must Copy Media Samples

Because every copy operation uses valuable CPU cycles, filter developers are encouraged to avoid copying the media samples, if possible. It is best to write the filter to modify media samples in place on an allocator acquired from another filter. In some cases, this is not possible, and a copy operation must be performed.

Where no copy is needed, the run-time overhead of a transform-inplace filter is scarcely more than that of a function; however, by packaging the transform as a filter, you get the full flexibility of the filter graph architecture.

Some reasons that a filter might be written as a copy transform filter rather than a transform-inplace filter include the following:

- If the transformation generates more data on output than space provided in the allocator of the input (for example, a decompressor filter), or if the transformation generates less data on output and must consolidate memory.
- If the original media data must be preserved; this is the case with the splitter, where the transform filter splits off a stream of data.
- If a decompressor filter is performing temporal compression, relying on adjacent frames for delta information. In this case, a separate copy must be made, primarily because the decompressor cannot allow another filter to have access to the original data to modify it.
- If the filter relies on a queue; for example, a filter that creates a queue to help smooth the delivery of irregularly spaced video frames would need to copy the samples.

Determine Which Base Classes to Use

Before choosing a base class for your transform filter, you must first decide whether your filter needs more than one input and output pin. If it does, you should derive your filter class from CBaseFilter.

If your filter needs to perform a video transform, you should derive your filter class from CVideoTransformFilter.

Otherwise, you should derive your filter class from CTransformFilter or CTransInPlaceFilter. To determine which one to use, you must decide whether your filter must copy media samples or can transform them in place. Because every copy operation uses valuable CPU cycles, filter developers should avoid copying media samples, if possible. It is best to write a filter to modify media samples in place on an allocator acquired from another filter. In some cases, this isn't possible, and you must perform a copy operation.

Where no copy is needed, the run-time overhead of a transform-inplace filter isn't much more than that of a function. However, by packaging the transform as a filter, you get the full flexibility of the filter graph architecture.

Some reasons that you might write a filter as a copy transform filter rather than a transform-inplace filter are:

- If the transformation generates more data on output than there is space already allocated for the input (for example, a decompressor filter), or if the transformation generates less data on output and must consolidate memory.

- If the original media data must be preserved. This is the case with the splitter, where the transform filter splits off a stream of data.
- If a decompressor filter is performing temporal compression, relying on adjacent frames for information about what has changed frame to frame. In this case, you must make a separate copy, primarily because the decompressor can't allow another filter to have access to the original data to modify it.
- If the filter relies on a queue; for example, a filter that creates a queue to help smooth the delivery of irregularly spaced video frames would need to copy the samples.

Once you determine whether the transform filter will copy media samples or transform them in place, you must decide which base class or classes to use and which member functions you must override and implement. You can then define your derived classes.

Some member functions in the base classes must be overridden in your derived class because they are either declared as pure virtual in the base classes (they have no implementation), or have default implementations that do nothing but return an error value.

You derive your filter class from the transform base classes CTransformFilter, CTransInPlaceFilter, or CVideoTransformFilter, or from the more generic CBaseFilter filter class. Most of the connection, media type, and allocator negotiation code is handled in the base classes and inherited by the transform classes. The transform classes make it possible to create a filter by deriving just one filter class (no pin classes). The transform classes make assumptions about the workings of transform filters that make the process of creating a transform filter easier.

To learn more about CTransformFilter and CTransInPlaceFilter and which of their member functions are typically overridden by the derived class, see Using the CTransformFilter and CTransInPlaceFilter Transform Base Classes.

Instantiate the Filter

All filters must add code to let the base classes instantiate the filter. To instantiate a filter, you must include two pieces of code in your filter: a static **CreateInstance** member function in the derived filter class, and a means of informing the class factory in the base classes how to access this function.

Typically, the **CreateInstance** member function calls the constructor for the derived filter class. The following is the **CreateInstance** member function from the Gargle sample filter.

```
CUnknown *CGargle::CreateInstance(LPUNKNOWN punk, HRESULT *phr) {

    CGargle *pNewObject = new CGargle(NAME("Gargle Filter"), punk, phr);
    if (pNewObject == NULL) {
        *phr = E_OUTOFMEMORY;
    }

    return pNewObject;
} // CreateInstance
```

To communicate with the class factory, declare a global array of CFactoryTemplate objects as g_Templates and provide the name of your filter, the class identifier (CLSID) of your filter, and a pointer to the static **CreateInstance** member function that creates your filter object. The Gargle sample filter does this as follows:

```
// Needed for the CreateInstance mechanism
CFactoryTemplate g_Templates[2]=
    { { L"Gargle filter"               , &CLSID_Gargle , CGargle::CreateInstance
    , { L"Gargle filter Property Page", &CLSID_GargProp, CGargleProperties::CreateI
    };

int g_cTemplates = sizeof(g_Templates)/sizeof(g_Templates[0]);
```

You can add additional parameters to the CFactoryTemplate templates if you want your filter to be self-registering. For more information on this, see Register DirectShow Objects.

Finally, link your filter to strmbase.lib and export DllGetClassObject and DllCanUnloadNow using a .def file.

Make Added Interfaces Available Through NonDelegatingQueryInterface

Only filters that add interfaces that are not in the base classes, such as those required for creating property pages, need implement the IUnknown member functions (called INonDelegatingUnknown in the base classes). The base classes provide default implementations of the **IUnknown** methods. **IUnknown** methods in any COM-based code retrieve interfaces from an object, and increment and decrement the reference counts of those interfaces. For example, the IUnknown::QueryInterface method retrieves interfaces from an object.

DirectShow defines a special IUnknown class called INonDelegatingUnknown, whose methods do the same thing as **IUnknown**. (The reason for the name change is so that objects can be aggregated.) The NonDelegatingQueryInterface method is called whenever some object or application wants to query a pin or filter for any interfaces it implements. If your filter implements any interface outside those listed in the base class implementation, you will need to override the **NonDelegatingQueryInterface** method to return a pointer to the implemented interface. For example, the following code example illustrates how the Gargle sample overrides the member function to distribute references to the ISpecifyPropertyPages and IPersistStream interfaces.

```
// Reveal our persistent stream, property pages, and IGargle interfaces
STDMETHODIMP CGargle::NonDelegatingQueryInterface(REFIID riid, void **ppv) {

    if (riid == IID_IGargle) {
        return GetInterface((IGargle *) this, ppv);
    } else if (riid == IID_ISpecifyPropertyPages) {
        return GetInterface((ISpecifyPropertyPages *) this, ppv);
    } else if (riid == IID_IPersistStream) {
        AddRef();     // Add a reference count to ourselves
        *ppv = (void *)(IPersistStream *)this;
        return NOERROR;

    } else {
        return CTransInPlaceFilter::NonDelegatingQueryInterface(riid, ppv);
    }
} // NonDelegatingQueryInterface
```

Note This sample calls the CTransInPlaceFilter implementation of the member function to finish up.

Override the Base Class Member Functions

When you determine which base class to use(see Determine Which Base Classes to Use), you write the header and define which member function to implement. You decide either to derive your filter class from the transform base classes (CTransformFilter or CTransInPlaceFilter), or from the more generic CBaseFilter filter class. In this section, you learn how to override the following member functions.

- Overriding the Transform Member Function
- Overriding the CheckInputType Member Function
- Overriding the CheckTransform Member Function
- Overriding the DecideBufferSize Member Function
- Overriding the GetMediaType Member Function
- Overriding Pin Member Functions
- Overriding the CBaseOutput::DecideAllocator Member Function

Overriding the Transform Member Function

The **Transform** member function in your derived class is called each time the IMemInputPin::Receive method on the input pin of the filter is called to transfer another sample. Place the code that performs the actual purpose of the filter in this member function, or in the functions called from here. Copy transform filters will likely have a private Copy member function associated with the transform code, while transform-inplace functions will simply modify the code in one buffer.

Overriding the CheckInputType Member Function

During the pin connection, the CheckMediaType member function of the input pin is called to determine whether the proposed media type is acceptable. The CTransformInputPin::CheckMediaType member function is implemented to call the CheckInputType member function of the derived filter class with the media type. You must implement this to accommodate the media types your filter can handle. The following code sample outlines part of the `CGargle::CheckInputType` member function, which rejects any media type but MEDIATYPE_Audio.

```
HRESULT CGargle::CheckInputType(const CMediaType *pmt) {
        ...
    // reject non-Audio type
    if (pmt->majortype != MEDIATYPE_Audio) {
        return E_INVALIDARG;
    }
```

Overriding the CheckTransform Member Function

Copy transform filters can transform the media type from the input pin to output pin. Therefore, if the output pin is connected (so its media type is known), when the CTransformInputPin::CheckMediaType member function is called during connection, the **CheckTransform** member function of the derived class is called to verify that the transform from the input type to the output type is valid. It is also called when CTransformOutputPin::CheckMediaType is called.

In the CTransInPlaceFilter class, this member function is implemented in the base class header file to simply return S_OK, because the functions from CTransformFilter that call this member function are overridden in **CTransInPlaceFilter** to call **CheckInputType** instead. This assumes that the media type doesn't change in a transform-inplace filter, as it might in a copy transform filter.

Overriding the DecideBufferSize Member Function

Copy transform filters might be required to set the properties of the allocator into which they are copying. This is likely if the downstream filter has provided a newly created allocator (that is, one that hasn't passed an allocator from farther downstream), or if the output pin is forced to create its own allocator. In this case, the pure virtual CBaseOutputPin::DecideBufferSize member function is called from the CBaseOutputPin::DecideAllocator member function, and the derived class fills in the requirements for the buffer by calling the IMemAllocator::SetProperties method on the allocator object to which it has a reference.

The CTransInPlaceFilter::DecideBufferSize method is never called, because the allocator of another filter is always in use. It is implemented in the base class header file to return E_UNEXPECTED.

Overriding the GetMediaType Member Function

Pins provide enumerators to enable other objects to determine the pin's media type. A pin provides the media type enumerator (the IEnumMediaTypes interface), which the pin base classes implement to call the **GetMediaType** member function in the pin class. In the copy transform filter classes, each pin's CTransformOutputPin::GetMediaType member function simply calls the virtual CTransformFilter::GetMediaType member function in the filter class. Your derived class must implement this member function to provide each supported media type in a list of media types.

In the transform-inplace classes, the enumerators form a transparent channel between the filters upstream and downstream from the transform filter. If the transform filter's input pin must perform an enumeration, it obtains an enumerator from the downstream filter's input pin. If the output pin must perform an enumeration, it obtains an enumerator from the upstream filter's output pin. One consequence of this is that transform-inplace filters can't connect to each other unless at least one of them is connected to something else, because neither of the transform-inplace filters can propose any media type for the connection.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

Overriding Pin Member Functions

If you derived your filter class from the transform classes and want more than one input or output pin, you must override the pin class (for example, CTransformInputPin or CTransformOutputPin). If you override the pin class, you must also override the **GetPin** member function of CTransformFilter or CTransInPlaceFilter, so that you can create pin objects from your derived classes. If you override one of the pin classes (for example,

CTransformInputPin) and override **GetPin** to create the pin object, you must also override **GetPin** to create the other pin object of the same base class (for example, **CTransformOutputPin**).

If you want more than one input or output pin, it is often simpler to derive your filter from CBaseFilter rather than from one of the transform classes.

Overriding the CBaseOutput::DecideAllocator Member Function

The base classes implement CBaseOutputPin::DecideAllocator to let the output pin automatically use the downstream pin's allocator. One of the most common alterations in the derived class is to force the use of an object's own allocator (or one from an upstream filter). In the DirectShow model, for example, a source filter pushes media samples onto the next filter and requires its own allocator. For example, if you write a transform-inplace filter and insert it between a source filter and a decompressor filter, the transform filter must present the source filter's allocator to the decompressor. Therefore, you must override the **CBaseOutputPin::DecideAllocator** member function.

Using the CTransformFilter and CTransInPlaceFilter Transform Base Classes

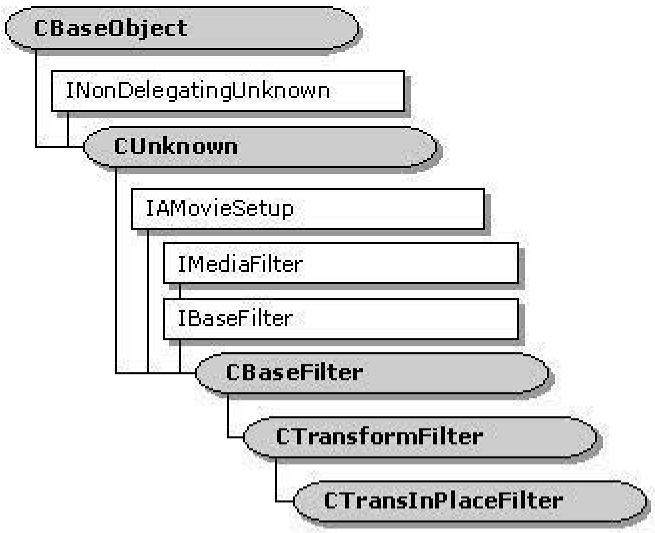
This article describes the classes provided for creating a transform filter. It is background information that you should read before reading the article Creating a Transform Filter, which walks through the steps of creating a transform filter.

Contents of this article:

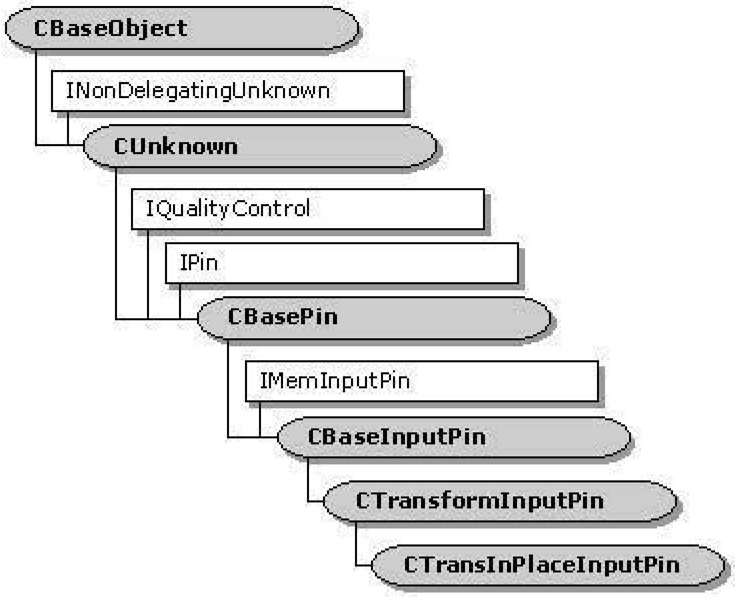
Introducing the CTransformFilter and CTransInPlaceFilter Classes

The easiest solution for writing a transform filter is to use the transform filter classes, which work well for most types of transform filters. Typically, a noncopying transform filter is derived from the CTransInPlaceFilter class and its associated pin classes; a copy transform filter is derived from the CTransformFilter filter class and its associated pin classes.

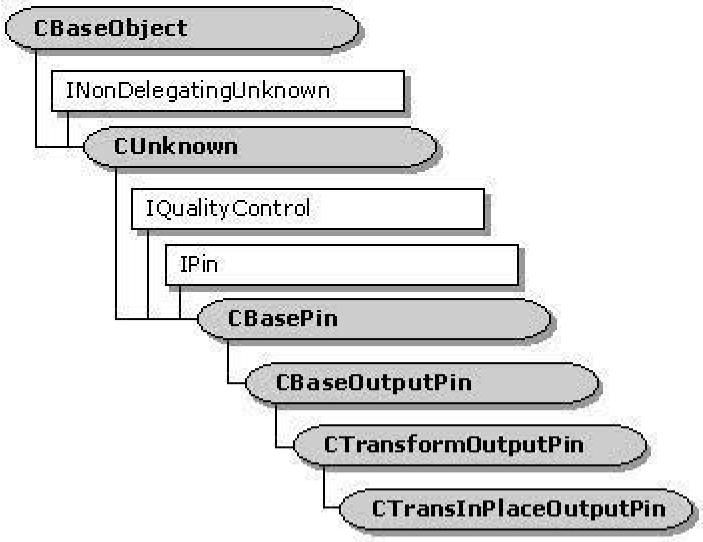
Transform filter classes are hierarchical, with the transform-inplace classes at the bottom of the hierarchy tree. CTransInPlaceFilter is derived from CTransformFilter, which is derived from CBaseFilter, as shown in the following illustration.



The CTransInPlaceInputPin class is derived from the CTransformInputPin class. The **CTransformInputPin** class is derived from the CBaseInputPin class, as shown in the following illustration.



The CTransInPlaceOutputPin class is derived from the CTransformOutputPin class. The **CTransformOutputPin** class is derived from the CBaseOutputPin class, as shown in the following illustration.



Copy transform and transform-inplace classes share many features, because the transform-inplace classes derive almost all member functions from the copy transform classes. The principal additions made by the transform classes over the base classes is that all required pin member functions are implemented—so for default implementation, you need only to derive a main filter class (from CTransInPlaceFilter or CTransformFilter).

What the Derived Class Must Provide

The derived filter class must provide a few member functions, typically to:

- Determine if the filter accepts the media type.
- Specify the count and size of any required allocators (for copy transforms only).
- Provide the transform functionality of the filter.

All derived filter classes must implement a static CFactoryTemplate::CreateInstance function. You can also choose to override the CBaseFilter::GetSetupData member function to make your filter self-registering. Beyond this, your classes must override a few member functions in the transform base classes. For more information about instantiating the filter, see Creating a Transform Filter.

If your derived filter class is based on the CTransformFilter class, you must override the following member functions.

Member function	Description
Transform	Implement transform.
CheckInputType	Verify support of media type.
CheckTransform	Verify support for transforming this type (for debugging builds only).
DecideBufferSize	Set size and count when copying.
GetMediaType	Suggest media types for the output pin.

If your derived filter class is based on the CTransInPlaceFilter class, override the following member functions.