Member function	Description
Transform	Implement transform.
CheckInputType	Verify support of media type.

Beyond providing your transform filter with a default implementation by providing the minimum overrides, you can override other member functions to provide more specialized behavior. Which member functions you override, of course, depends on what you want your filter to do. For example, you must override the GetPin and GetPinCount member functions if you want to have more than one input pin and one output pin on the filter.

Also, several base class member functions, such as BreakConnect or CompleteConnect, are called as notifications to your filter through the pins. Typically, most of these member functions exist only on the pins. In the classes based on CTransformFilter, the pin functions are implemented to call similarly named functions in the filter class. This means that the member functions you most likely will want to override are all collected into one filter class, so you can leave the pin classes unchanged, making implementation smaller and easier. These member functions are as follows:

Member function	Reason to override
NonDelegatingQueryInterface	To distribute any interfaces added in the derived class.
GetPinCount	If adding more pins to the transform filter.
GetPin	If adding more pins to the transform filter.
CheckConnect	To obtain extra interfaces at connect time or for other reasons.
BreakConnect	To release extra interfaces when connection is broken or for other reasons.
CompleteConnect	To perform some action at the end of connection (such as reconnecting the input pin).
SetMediaType	To be notified when the media type has been set.
StartStreaming	To be notified when entering the streaming state.
StopStreaming	To be notified when exiting the streaming state.
AlterQuality	To do anything with quality-control messages other than passing them on.

A Sample Transform Filter Declaration

An example of a filter derived from a transform class is the NullNull sample filter. This sample illustrates a true minimalist filter, which does nothing except demonstrate the least you must implement for a filter. It uses the transform-inplace classes and derives its filter class from the CTransInPlaceFilter class. Following is the class declaration for the derived filter class CNullNull.

```
// CNullNull
//
class CNullNull
    : public CTransInPlaceFilter
{

public:

    static CUnknown *CreateInstance(LPUNKNOWN punk, HRESULT *phr);

    DECLARE_IUNKNOWN;
    LPAMOVIESETUP_FILTER GetSetupData()
    {
        return &sudNullNull;
    }

private:

    // Constructor - just calls the base class constructor
    CNullNull(TCHAR *tszName, LPUNKNOWN punk, HRESULT *phr)
        : CTransInPlaceFilter (tszName, punk, CLSID_NullNull, phr)
    { }

    // Overrides the PURE virtual Transform of CTransInPlaceFilter base class
    // This is where the "real work" is done by altering *pSample.
    // We do the Null transform by leaving it alone.
    HRESULT Transform(IMediaSample *pSample){ return NOERROR; }

    // We accept any input type.  We'd return S_FALSE for any we didn't like.
    HRESULT CheckInputType(const CMediaType* mtIn) { return S_OK; }
};
```

This example illustrates the basic member functions required in the base class:

CreateInstance	Needed by every filter so that it can be instantiated as a COM object.
GetSetupData	Overrides CBaseFilter::GetSetupData and is used to provide the class with information required to register this particular filter. In this case, it provides the address of a structure defined in the Nullnull.cpp file included in the SDK.
CNullNull	Class constructor, which typically just calls the base class constructor.
Transform	Overrides CTransInPlaceFilter::Transform and does the main work of CNullNull, which in this case is nothing.
CheckInputType	Overrides CTransInPlaceFilter::CheckInputType to verify the media type during connection, and in this case accepts any media type offered, since it will simply pass it along to the next filter in line.

Note that, strictly speaking, GetSetupData is required only if you want your filter to be self-registering. However, since the base classes implement this feature and it is easy to implement, it is a good idea to include this in your base class.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

Connecting Transform Filters

This article describes some of the connection issues faced when creating a transform filter. Connecting any two filters requires negotiating which media types to use and deciding on a common allocator for passing samples. Since transform filters are connected on both sides, and since some transform filters use the media types and allocators of other filters in the graph, it is important to understand the concepts involved in transform filter connections.

Contents of this article:

How Allocator Negotiation Works

For information about the connection process, including media type and allocator negotiation, see Connection Model. When you are determining your transform filter characteristics, it might help for you to understand the model of allocator negotiation for transform filters.

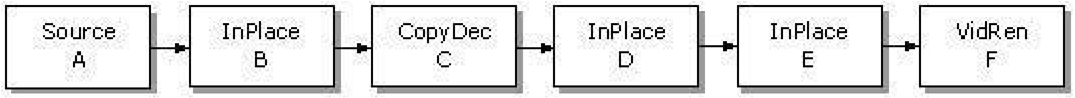
If your transform filter requires copying, then it will copy media samples from a buffer established by its input pin to a buffer established by its output pin. These buffers are provided by allocators that might actually be located in other filters, perhaps even several filters removed if the filters in between do not copy the data.

A copying transform filter typically tries to use the allocator of the upstream filter for its input pin, and the allocator of the downstream filter for its output pin. During the connection process, the output pin of the upstream filter determines which allocator to use for the upstream transport, so the input pin of the copy transform must be prepared to create an allocator for the upstream transport if its IMemInputPin::GetAllocator method is called by the connecting output pin. The base classes provide a way to create a new allocator from the input pin of any connection.

On the other hand, transform-inplace filters do not make copies, but rather modify the data in an existing buffer. These filters should always offer the allocator from the downstream filter to the upstream filter. This requires a reconnection, because the filter does not know about the downstream filter when its input pin is first connected. Also, because inplace-transforms do not change the media type, the media type from the downstream filter should be offered to the upstream filter upon reconnection.

Connecting Filter Graphs: An Example

To better understand the allocation model for a transform-inplace filter, the following illustration shows a simple and common example of a filter graph.



This simple example demonstrates the model of the transform-inplace filter offering its downstream allocator to its upstream filter. Consider what happens when InPlace E is connected to VidRen F.

Upon connection, the video renderer filter (VidRen F) offers its allocator for use by the upstream inplace filter (InPlace E). Because it is a transform-inplace filter, InPlace E offers the allocator to the next filter upstream, InPlace D, and so on. This reconnection and renegotiation occurs until it encounters either the source filter or a copy transform filter. In this case it meets a decompressor, CopyDec C. (The copy transform filter cannot offer its allocator upstream, because it must perform a copy operation.) So the decompressor will be writing directly to the video renderer's buffer, which might be a DirectDraw® surface. This demonstrates why it is a good practice to write a transform filter as a transform-inplace filter and pass allocators from the renderer upstream, if possible.

On the other hand, consider filters InPlace B and CopyDec C. What if the downstream filter from a transform-inplace filter is a copy transform filter instead of a renderer? In this case, the copy transform filter will offer to create its own allocator on its input pin (the base classes handle this), and the transform-inplace filter can then offer that allocator downstream upon reconnection (the same as if it were connected to a renderer filter).

However, even though CopyDec C can create its own allocator (from its IMemInputPin::GetAllocator method), the source filter, Source A, uses its own buffer—say, a file. So when InPlace B connects to CopyDec C, InPlace B will have accepted the source filter's allocator already and will force that allocator to be used for the transport between itself and the decompressor filter. InPlace B can then provide the upstream filter, Source A, with the option of using the allocator offered by CopyDec C, but the source filter will refuse this allocator so that an extra copy does not have to be made from the file buffer to the decompressor's input buffer.

Therefore, any upstream filter can force the use of its allocator downstream but should have good reason to do so (such as if it already has a buffer). In this example, only one copy is being made (by the decompression filter) between the file buffer and the video memory.

Establishing Media Type Connections

When pins from different filters are connected, they must both agree on a common media type for the samples they will exchange. A quick review of the connection mechanism might help highlight how transform filters handle media type negotiations.

This section contains the following topics.

- The Connection Process: A Summary
- When Reconnections Occur

The Connection Process: A Summary

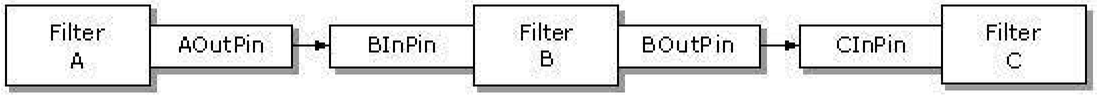
When one filter's output pin is called by the filter graph manager to connect to an input pin of a second filter, the IPin::Connect method is called. This, in turn, calls CBaseOutputPin::CheckConnect to obtain interfaces from the connected pin and CBasePin::AgreeMediaType to find a common media type.

AgreeMediaType calls CBasePin::TryMediaTypes, which uses media type enumerators to query the pins for preferred media types. IEnumMediaTypes is an interface on the connected input pin that TryMediaTypes uses first. The base classes use **IEnumMediaTypes** to repeatedly call a CBasePin member function called GetMediaType for each media type in the list. You use this member function in your input and output pin classes to return the media types that your pin prefers.

TryMediaTypes calls the output pin's CheckMediaType member function with each input type returned. You must use **CheckMediaType** to verify whether this type is acceptable. If no media types are found (for example GetMediaType may not even be used on the connected input pin, or may return an unacceptable media type), then AgreeMediaType obtains a media type enumerator for the output pin and tries each of these in turn. Again, the **GetMediaType** member function of the derived output pin is called for each type. In this case, it can determine media types by inquiring about any existing connection established by the filter's input pin.

When Reconnections Occur

For transform filters that do not modify the media type from input pin to output pin (such as most in-place transforms and many copy transforms), a reconnection scheme must be in place for offering the downstream filter's media type to the upstream filter. To understand this, consider the media type negotiation of the transform-inplace Filter B in the following illustration.



The input pin of Filter B is connected first and establishes a media type with the upstream output pin (AOutPin). When the output pin of Filter B is connected next, it must use the enumerator from the output pin of the connected upstream filter (AOutPin), because it does not have one of its own.

If the pin of the downstream filter, CInPin, can accept this, then the connection is complete. However, assume that Filter C does not agree to this media type but proposes a media type that Filter B can handle.

Before deciding that it can handle the media type, Filter B calls the IPin::QueryAccept method on AOutPin to ensure that it is acceptable. If no media type can be found that is acceptable for all the filters, then the BOutPin to CInPin connection will fail. (It is possible to find that a transform-inplace filter will connect to either its upstream or its downstream neighbors, but not both simultaneously.)

If a suitable type is found, BOutPin must force a reconnection on the entire filter, and pass the established media type (the media type of CInPin) to AOutPin, when AOutPin and BInPin are connected again.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

About Compression Filters

A *compression filter* is a specialized type of transform filter. Compression filters (compressors) accept data, use a compression scheme to transform the data, and pass the compressed data downstream.

Microsoft® DirectShow™ includes an AVI Compressor filter and an ACM Audio Compressor filter, which will use any Microsoft Video for Windows® video or audio codec to compress data. You can write your own compressor filter if you need to compress data in a format that isn't supported by the default filters that DirectShow provides.

To begin writing a compression filter, write a transform filter that includes one input pin and one output pin. See the following articles for more information about writing a transform filter.

- Creating a Transform Filter
- Using the CTransformFilter and CTransInPlaceFilter Transform Base Classes
- Connecting Transform Filters

After you've written a transform filter, you should review the following points when completing your compression filter:

- Register your filter.

  Register your compression filter by using the AMovieDllRegisterServer2 function. This enables applications to enumerate your filter with all the other compression filters on the system. See Enumerate and Access Hardware Devices in DirectShow Applications for more information about device enumeration.

- Implement the recommended compressor interfaces.

  It is strongly recommended that you implement the IAMStreamConfig interface on the output pin of all compression filters and IAMVideoCompression on the output pin of video compressors so that applications can access the compression features of your filter.

  IAMStreamConfig enables you to inform applications about the formats to which you can compress data, and enables the application to configure your compressor to compress to a particular data type.

  IAMVideoCompression enables an application set video-specific settings, like keyframe frequency, that do not appear in the AM_MEDIA_TYPE structure.

The VidCap Sample (Video Capture Filter) sample video capture filter included with the DirectShow SDK implements the IAMStreamConfig and IAMVideoCompression interfaces, and performs filter registration. Note that this sample code is for a capture filter, but the filter registration and implementation of these two interfaces is similar to that of a compression filter.

Previous Home Topic Contents Index Next

About Effect Filters

In DirectShow, *effect filters* are defined as filters that apply an effect to media data, but don't change the media type. DirectShow provides several effect filters, including Contrast, Gargle, and EzRGB24. Effect filters can apply a wide range of useful video and audio effects to media data.

Contents of this article:

Creating Effect Filters

Because the input and output media formats are the same, and the applied effect can't change the format, effect filters often contain a code that checks the media formatting. If the filter derives from one of the transform filter base classes, CTransformFilter or CTransInPlaceFilter, the filter typically checks the format with the CheckMediaType, CheckInputType, and CheckTransform methods. If the filter doesn't derive from one of the transform filter base classes, its pins typically check the format by calling the CBasePin::CheckMediaType member function. See Negotiating Media Types with CBasePin::AgreeMediaType for more information.

You should choose a base class for your effect filter class that provides the greatest amount of the functionality you need. Often, the base class will be one of the transform filter base classes. If none of the higher-level base classes support your required functionality, you can choose CBaseFilter or CBasePin as your base class.

Your effect filter must implement the IPersistStream interface if you want to save the state of your effects in the Filter Graph Editor. To access this interface, derive your effect filter class from CPersistStream and query for the **IPersistStream** interface. Saving the filter's state can be helpful during design, but it is often useful to have the effect filter return to a default state when the Filter Graph Editor closes it, in which case you don't need to implement **IPersistStream**.

If you want the user to be able to manipulate the effect, you must create and display your effect filter's property page and provide a mechanism for returning the user's input to the filter. To do this, implement a property page class, the ISpecifyPropertyPages interface (which exposes property pages), and a custom interface that changes property page values. Typically, property pages use controls such as a slider, button, or check box to receive user input. You also must provide the resource file that displays the controls on the property page.

To implement the property page class, create a class that derives from CBasePropertyPage and implement the OnReceiveMessage method, the CPersistStream::SetDirty method, and a data member for each effect parameter. To access the two interfaces, derive your effect filter class from ISpecifyPropertyPages and the custom interface, and then query for the interfaces. You can query for all the interfaces you need by overriding the NonDelegatingQueryInterface method as shown in the following code from the Gargle sample (IGargle is the custom interface):

```
STDMETHODIMP CGargle::NonDelegatingQueryInterface(REFIID riid, void **ppv)
{
    CheckPointer(ppv,E_POINTER);
    if (riid == IID_IGargle) {
        return GetInterface((IGargle *) this, ppv);
    } else if (riid == IID_ISpecifyPropertyPages) {
        return GetInterface((ISpecifyPropertyPages *) this, ppv);
    } else if (riid == IID_IPersistStream) {
        return GetInterface((IPersistStream *) this, ppv);
    } else {
        return CTransInPlaceFilter::NonDelegatingQueryInterface(riid, ppv);
    }
}
```

The effect filter's custom interface typically supplies a put and a get method for each effect parameter. For example, the IGargle custom interface supplies put_GargleRate and get_GargleRate methods. The IContrast custom interface in the Contrast sample supplies put_ContrastLevel and get_ContrastLevel methods. When the user accesses one of the controls on the property page, the page generates a windows message. The property page class's OnReceiveMessage member function handles this message. The following code fragment from the Contrast sample demonstrates this message generation and handling. IDB_DEFAULT is the resource ID of the Default button. The user clicks this button to set the video contrast to its default state. The CContrastProperties class implements the property page and the IContrast::put_DefaultContrastLevel method sets the contrast level to its default value.

```
BOOL CContrastProperties::OnReceiveMessage(HWND hwnd, UINT uMsg,
                                WPARAM wParam, LPARAM lParam)
{
    switch (uMsg)
    {
       case WM_COMMAND:
        {
          if (LOWORD(wParam) == IDB_DEFAULT)
            {
                pIContrast()->put_DefaultContrastLevel();
                SendMessage(m_hwndSlider, TBM_SETPOS, TRUE, 0L);
              SetDirty();
            }
            return (LRESULT) 1;
        }
...
```

Effect filters use critical sections internally to protect the global filter state. Effect filters can lock a critical section to ensure that data flow through the filter graph is serialized and that the global filter state doesn't change while an effect is occurring. DirectShow locks a critical section by declaring a CAutoLock class object. Typically, effect filters lock the critical section as soon as they enter the function that applies the effect. For example, in the following code fragment from the Gargle sample, the function that applies the effect is MessItAbout:

```
CCritSec   m_GargleLock; // Declare the critical section data member in the effect

void CGargle::MessItAbout(PBYTE pb, int cb)
{
    CAutoLock foo(&m_GargleLock);
```

The put and get methods of the effect properties (for example, put_GargleRate) typically lock the critical section so effect values can't change in the middle of an update.

List of DirectShow Effect Filters and Samples

The DirectShow SDK ships with the following effect filters. You can find these filters in the Samples directory. All the source code is included.

Contrast: This effect filter adjusts the contrast of the video images sent through it. The filter adjusts the contrast by using palettes, because an image's color palette effectively determines how the image is interpreted by the display device; that is, how the value 23 (for example) maps into an RGB triplet for display. By changing the palette, you can reduce and increase contrast without doing anything to the image pixels themselves. The Filter Graph Editor lists this filter as Video Contrast.

EzRGB24: This effect filter modifies decompressed video images sent through it. It creates color and image filtering effects through simple techniques such as adjusting the red, green, or blue levels (to change the playback color) and by averaging neighboring pixels (to achieve blur and embossed (raised) effects). The Filter Graph Editor lists this filter as Image Effects.

Gargle: This effect filter modifies audio data sent through it. A synthesized wave function modulates the audio data's amplitude. The secondary wave can be a triangular or square wave, and can have different frequencies. At low modulation frequencies it sounds like a tremolo. At high modulation frequencies it sounds like a distortion. The Filter Graph Editor lists this filter as Gargle.

Video Renderers

This section describes how to write and use video renderers, both full-screen and custom renderers. It discusses how and why to support a full-screen renderer, and how to handle notifications, state changes, and dynamic format changes in a custom renderer.

Full-Screen Video Renderer

This article explains the logic used by the default Microsoft® IVideoWindow plug-in distributor (PID), when an application instructs it to render the video in full-screen mode. Substitute renderer filters can use the **IVideoWindow** PID for communication with applications. Developers of substitute renderers should be aware of how this PID searches the filter graph for the best means of representing full-screen video, when requested to render in full-screen mode.

Contents of this article:

Using the IVideoWindow PID to Implement Full-Screen Support

Microsoft® DirectShow™ implements full-screen support in a number of ways that depend on what hardware resources are available. An application can support full-screen video playback through the IVideoWindow interface provided by the filter graph manager. An application can have its own implementation of full-screen playback, but it can probably make better use of resources by using the **IVideoWindow** implementation.

The IVideoWindow plug-in distributor (PID) tries three different options for implementing full-screen support when an application requests full-screen mode. The option is typically chosen the first time the filter graph enters full-screen mode. While in full-screen mode, no **IVideoWindow** methods can be called (apart from accessing the full-screen property). Any attempts to do so will return the VFW_E_IN_FULLSCREEN_MODE message. The PID searches in the following order for a filter that supports **IVideoWindow** and that has one of these characteristics:

1. The filter supplies full-screen mode directly.
2. The filter allows its window to be stretched to full screen without penalty.
3. The filter can be replaced by a full-screen renderer.

If none of these three options are found, the default is to simply stretch the video of a filter that supports IVideoWindow to full-screen, ignoring the performance penalties.

Finding a Filter That Supports Full-Screen Mode

The first option is to search for a filter in the filter graph that supports full-screen mode directly. When asked to go into full-screen mode, the IVideoWindow PID first scans all filters supporting **IVideoWindow** in the filter graph. The PID calls IVideoWindow::get_FullScreenMode on each filter and, if the filter returns E_NOTIMP (the default), assumes that the filter has no inherent support for full-screen playback. If the filter returns anything else, then that filter becomes the nominated filter for full-screen playback. This means that any calls to the filter graph manager to set the full-screen mode on or off will be sent directly to that nominated filter. This mechanism allows filters to be extended to support full-screen support directly. Most normal window-based renderers do not need to support this feature.

Finding a Filter That Can Be Stretched Full-Screen at No Cost

If a full-screen rendering filter can't be found, then the PID tries to find a filter supporting the IVideoWindow interface that can have its window stretched full-screen without penalty. The PID does this by scanning the list of filters in the filter graph that support **IVideoWindow**. For each filter found, the PID calls IVideoWindow::GetMaxIdealImageSize. If a filter indicates that its window can be stretched full-screen at no cost, then that becomes the nominated filter.

If that nominated filter is then requested to render full-screen, the PID resets a number of the filter's IVideoWindow properties and stretches the window to full-screen. This typically means setting a null owner, changing the window styles to not show the border or the caption, and updating the window position to match the current display size. When full-screen mode is switched off, the properties on the filter will be reset to the state prior to the full-screen mode.

Most video renderers supporting IVideoWindow cannot return the maximum ideal image size until they have been activated (either paused or running), because that is when they allocate their resources. For this reason, when scanning the list of filters while the filter graph is in a stopped state, the PID pauses each filter before calling IVideoWindow::GetMaxIdealImageSize. After calling this method, the filter is stopped again.

Supplying a Full-Screen Renderer Filter

If neither of the previous options were successful, then the PID finds the first available filter in the filter graph that supports an IVideoWindow interface, and assumes that it is the current video renderer filter.

If no filters that support IVideoWindow are available, the call to change to full-screen mode will return VFW_E_NO_FULLSCREEN. When asked to change into full-screen mode, the PID stops the filter graph, if it is not already stopped, disconnects the current renderer, and reconnects the DirectShow full-screen renderer in its place. If the connection succeeds, then the filter graph is restored to its original state. When switching out of full-screen mode, the opposite occurs. That is, the full-screen renderer is disconnected and the original filter is reconnected. The state of the filter graph is likewise restored. The full-screen renderer is a specialized renderer that uses the display changing capabilities provided by DirectDraw®. For example, it might switch the full-screen display 320 × 240 when it might have been in, for example, 1024 × 768. By switching to lower resolution modes, it can cheaply implement full-screen rendering without having to stretch images.

The full-screen renderer currently implements 320 × 200 × 8/16 bits per pixel, 320 × 240 × 8/16, 640 × 400 × 8/16, 640 × 400 × 8/16, 640 × 480 × 8/16, 800 × 600 × 8/16, 1024 × 768 × 8/16, 1152 × 864 × 8/16, and 1280 × 1024 × 8/16 display modes. The Modex renderer supports the IFullScreenVideo interface. When the modex renderer is connected, it loads the display modes DirectDraw has made available. The number of modes available can be obtained through IFullScreenVideo::CountModes. Information on each mode is available by calling IFullScreenVideo::GetModeInfo and IFullScreenVideo::IsModeAvailable. An application can enable and disable any modes by calling the SetEnabled flag with OATRUE or OAFALSE. The current value can be queried for with IFullScreenVideo::IsModeEnabled.

Another way to set the modes enabled is to use the clip loss factor. This defines the amount of video that can be lost when deciding which display mode to use. Assuming the decoder cannot compress the video, then playing, for example, an MPEG file that is 352 × 288 pixels into a 320 × 200 display will lose over 40 percent of the image. The clip loss factor specifies the upper range of clipping loss that is permissible. To allow typical MPEG video to be played in 320 × 200 it defaults to 50 percent. You can set the clip loss factor with IFullScreenVideo::SetClipFactor.

Stretching the Output of a Renderer Full-Screen

After trying the first three options, the final option for implementing full-screen support is to pick any filter enabled by IVideoWindow and stretch its window full-screen, regardless of the resulting poor performance. Essentially, the first filter in the filter graph that is enabled by the **IVideoWindow** interface becomes the nominated filter. This filter is then used in the same manner as if it was a filter that could be stretched full-screen without sacrificing performance (that is, the owner is reset, the styles changed, and the window position changed to match the display extents).

The cost of stretching a window full-screen where there is an implicit performance penalty varies, depending on the resolution currently displayed. The worst scenario is one in which the user is using a relatively high resolution (for example, 1024 × 768) and the images must be stretched by the renderer using GDI. This is likely to provide very low frame throughput and is used only as a last resort.

Implications of Full-Screen Support for the Application

While the interface exposed to applications is relatively simple, the underlying implementation can be more complex. The full-screen renderer has some special properties that application developers should be aware of. In particular, the renderer changes display modes only when activated (either paused or running). Therefore, if the filter graph is stopped when switching to full-screen mode, no change might be obviously visible until the filter graph is started again. When the filter graph is subsequently run, the display mode will change and the full-screen rendering will start.

If a window is being stretched full-screen (that is, no full-screen renderer is being used), the change will be viewable when the full-screen mode is set, regardless of state. If full-screen playback is being supported directly by a filter in the filter graph, it might elect to copy the behavior of the full-screen renderer and switch to full-screen only when activated. The filter supporting full-screen playback might have to do this, because the resources they require to play full-screen might not be available until then. Therefore, an application should avoid setting full-screen mode when stopped.

This makes sense in a user interface context as well, because if full-screen mode is set when the filter graph is stopped, users are unlikely to be able to start the graph running without switching out of full-screen mode (that is, tabbing back to the original application).

All renderers that implement IVideoWindow send event codes to the filter graph manager when their windows are activated or deactivated. When in full-screen mode, the PID watches for these event codes. When it sees a window that it made full-screen being deactivated, it will automatically switch out of full-screen mode and send an EC_FULLSCREEN_LOST notification to the application event queue. This is the only interference caused by the PID; all other user interface is left open to the application as described in the remainder of this article.

One of the most important aspects of full-screen playback is that when in full-screen mode, no window can be displayed on top of the full-screen window. In fact, when the full-screen renderer switches display modes, it disables all GDI output for other applications, so displaying a window on top of a full-screen window is actually impossible. Any user interactions with the computer must be done through hot keys.

Whatever mechanism the PID ultimately uses to implement full-screen playback, it always ensures that the message drain property is set on the window executing the playback. (The message drain specifies a window that will be forwarded all Windows® messages sent to the renderer.) So, even if the full-screen renderer is used, as long as a message drain has previously been set on the filter graph manager's IVideoWindow interface, all messages will be passed on to that renderer.

Because the message drain is set on the appropriate window, an application can rely on receiving all mouse and keyboard messages when in full-screen mode, regardless of which filter is implementing it. An application can use this fact to implement hot-key support for seeking, for example. However, properties can be set only when not in full-screen mode, so if the only time an application is required to catch messages is when it is in full-screen mode, it must set the message drain before setting full-screen on. Likewise, the message drain can be reset only after setting full-screen mode off.

One other application consideration is that, when in full-screen mode, any source and destination rectangles set through IBasicVideo will not be adhered to. The PID resets these rectangles when switching to full-screen mode. It does this because not all filters implementing full-screen support can guarantee to support **IBasicVideo** as well.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

Alternative Video Renderers

This article describes some of the more complicated implementation requirements of a renderer; these apply to most renderers, although some aspects are video-specific (such as EC_REPAINT and other notifications). In particular, it discusses how to handle various notifications, state changes, and format changes. It also provides a summary of the notifications that a renderer is responsible for sending to the filter graph manager.

Contents of this article:

Writing an Alternative Renderer

Microsoft® DirectShow™ provides a window-based video renderer; it also provides a full-screen renderer in the run-time installation. You can use the C++ classes in the DirectShow SDK to write alternative video renderers. For alternative renderers to interact correctly with DirectShow-based applications, the renderers must adhere to the guidelines outlined in this article. You can use the CBaseRenderer and CBaseVideoRenderer classes to help follow these guidelines when implementing an alternative video render. Consult the SampVid sample in the DirectShow SDK for an example of an alternative video renderer that uses these classes. Because of the ongoing development of DirectShow, review your implementation periodically to ensure that the renderers are compatible with the most recent version of DirectShow.

This article discusses many notifications that a renderer is responsible for handling. A brief review of DirectShow notifications might help to set the stage. There are essentially three kinds of notifications that occur in DirectShow:

- *Stream notifications*, which are events that occur in the media stream and are passed from one filter to the next. These can be begin-flushing, end-flushing or end-of-stream notifications and are sent by calling the appropriate method on the downstream filter's input pin (for example IPin::BeginFlush).
- *Filter graph manager notifications*, which are events sent from a filter to the filter graph manager such as EC_COMPLETE. This is accomplished by calling the IMediaEventSink::Notify method on the filter graph manager.
- *Application notifications*, which are retrieved from the filter graph manager by the controlling application. An application calls the IMediaEvent::GetEvent method on the filter graph manager to retrieve these events. Often, the filter graph manager passes through the events it receives to the application.

This article discusses the responsibility of the renderer filter in handling stream notifications it receives and in sending appropriate filter graph manager notifications.

Handling End-of-stream and Flushing Notifications

An end-of-stream notification begins at an upstream filter (such as the source filter) when that filter detects that it can send no more data. It is passed through every filter in the graph and eventually ends at the renderer, which is responsible for subsequently sending an EC_COMPLETE notification to the filter graph manager. Renderers have special responsibilities when it comes to handling these notifications.

A renderer receives an end-of-stream notification when its input pin's IPin::EndOfStream method is called by the upstream filter. A renderer should note this notification and continue to render any data it has already received. Once all remaining data has been received, the renderer should send an EC_COMPLETE notification to the filter graph manager. The **EC_COMPLETE** notification should be sent only once by a renderer each time it reaches the end of a stream. Furthermore, **EC_COMPLETE** notifications must never be sent except when the filter graph is running. Therefore, if the filter graph is paused when a source filter sends an end-of-stream notification, then **EC_COMPLETE** should not be sent until the filter graph is finally run.

Any calls to the IMemInputPin::Receive or IMemInputPin::ReceiveMultiple methods after an end-of-stream notification is signaled should be rejected. E_UNEXPECTED is the most appropriate error message to return in this case.

When a filter graph is stopped, any cached end-of-stream notification should be cleared and not resent when next started. This is because the filter graph manager always pauses all filters just before running them so that proper flushing occurs. So, for example, if the filter graph is paused and an end-of-stream notification is received, and then the filter graph is stopped, the renderer should not send an EC_COMPLETE notification when it is subsequently run. If no seeks have occurred, the source filter will automatically send another end-of-stream notification during the pause state that precedes a run state. If, on the other hand, a seek has occurred while the filter graph is stopped, then the source filter might have data to send, so it won't send an end-of-stream notification.

Video renderers often depend on end-of-stream notifications for more than the sending of EC_COMPLETE notifications. For example, if a stream has finished playing (that is, an end-of-stream notification is sent) and another window is dragged over a video renderer window, a number of WM_PAINT window messages will be generated. The typical practice for running video renderers is to refrain from repainting the current frame upon receipt of WM_PAINT messages (based on the assumption that another frame to be drawn will be received). However, when the end-of-stream notification has been sent, the renderer is in a waiting state; it is still running but is aware that it will not receive any additional data. Under these circumstances, the renderer customarily draws the playback area black.

Handling flushing is an additional complication for renderers. Flushing is carried out through a pair of IPin methods called BeginFlush and EndFlush. Flushing is essentially an additional state that the renderer must handle. It is illegal for a source filter to call **BeginFlush** without calling **EndFlush**, so hopefully the state is short and discrete; however, the renderer must correctly handle data or notifications it receives during the flush transition.

Any data received after calling BeginFlush should be rejected immediately by returning E_UNEXPECTED. Furthermore, any cached end-of-stream notification should also be cleared when a renderer is flushed. A renderer will typically be flushed in response to a seek. The flush ensures that old data is cleared from the filter graph before fresh samples are sent. (Typically, the playing of two sections of a stream, one after another, is best handled through deferred commands rather than waiting for one section to finish and then issuing a seek command.)

Handling State Changes and Pause Completion

A renderer filter behaves the same as any other filter in the filter graph when its state is changed, with the following exception. After being paused, the renderer will have some data queued, ready to be rendered when subsequently run. When the video renderer is stopped, it holds on to this queued data. This is an exception to the DirectShow rule that no resources should be held by filters while the filter graph is stopped.

The reason for this exception is that by holding resources, the renderer will always have an image with which to repaint the window if it receives a WM_PAINT message. It also has an image to satisfy methods, such as CBaseControlVideo::GetStaticImage, that request a copy of the current image. Another effect of holding resources is that holding on to the image stops the allocator from being decommitted, which in turn makes the next state change occur much faster because the image buffers are already allocated.

A video renderer should render and release samples only while running. While paused, the filter might render them (for example, when drawing a static poster image in a window), but should not release them. Audio renderers will do no rendering while paused (although they may perform other activities, such as preparing the wave device, for example). The time at which the samples should be rendered is obtained by combining the stream time in the sample with the reference time passed as a parameter to the IMediaControl::Run method. Renderers should reject samples with start times less than or equal to end times.

When an application pauses a filter graph, the filter graph does not return from its IMediaControl::Pause method until there is data queued at the renderers. In order to ensure this, when a renderer is paused, it should return S_FALSE if there is no data waiting to be rendered. If it has data queued, then it can return S_OK.

The filter graph manager checks all return values when pausing a filter graph, to ensure that the renderers have data queued. If one or more filters are not ready, then the filter graph manager polls the filters in the graph by calling GetState. The **GetState** method takes a time-out parameter. A filter (typically a renderer) that is still waiting for data to arrive before completing the state change returns VFW_S_STATE_INTERMEDIATE if the **GetState** method expires. Once data arrives at the renderer, **GetState** should be returned immediately with S_OK.

In both the intermediate and completed state, the reported filter state will be State_Paused. Only the return value indicates whether the filter is really ready or not. If, while a renderer is waiting for data to arrive, its source filter sends an end-of-stream notification, then that should also complete the state change.

Once all filters actually have data waiting to be rendered, the filter graph will complete its pause state change.

Handling Termination

Video renderers must correctly handle termination events from the user. This implies correctly hiding the window and knowing what to do if a window is subsequently forced to be displayed. Also, video renderers must notify the filter graph manager when its window is destroyed (or more accurately, when the renderer is removed from the filter graph) to free resources.

If the user closes the video window (for instance by pressing ALT+F4), the convention is to hide the window immediately and send an EC_USERABORT notification to the filter graph manager. This notification is passed through to the application, which will stop the graph playing. After sending **EC_USERABORT**, a video renderer should reject any additional samples delivered to it.

The abort flag should be left on by the renderer until it is subsequently stopped, at which point it should be reset so that an application can override the user action and continue playing the graph if it desires. If ALT+F4 is pressed while the video is running, the window will be hidden and all further samples delivered will be rejected. If the window is subsequently shown (perhaps through IVideoWindow::put_Visible), then no EC_REPAINT notifications should be generated.

The video renderer should also send the EC_WINDOW_DESTROYED notification to the filter graph when the video renderer is terminating. In fact, it is best to handle this when the renderer's IBaseFilter::JoinFilterGraph method is called with a null parameter (indicating that the renderer is about to be removed from the filter graph), rather than waiting until the actual video window is destroyed. Sending this notification allows the plug-in distributor in the filter graph manager to pass on resources that depend on window focus to other filters (such as audio devices).

Handling Dynamic Format Changes

Video renderers in DirectShow accept only video formats that can be drawn efficiently. For example, the window-based run-time renderer will accept only the RGB format that matches the current display device mode (for example, RGB565 when the display is set to 65,536 colors). As a last resort, it also accepts 8-bit palettized formats, as most display cards can draw this format efficiently. When the renderer has Microsoft® DirectDraw® loaded, it might later ask the source filter to switch to something that can be written onto a DirectDraw surface and drawn directly through display hardware. In some cases, the renderer's upstream filter might try to change the video format while the video is playing. This often occurs when a video stream has a palette change. It is most often the video decompressor that initiates a dynamic format change.

An upstream filter attempting to change formats dynamically should always call the IPin::QueryAccept method on the renderer input pin (for filters based on CTransformFilter, this is implemented in CTransformFilter::CheckInputType). It is undefined as to which formats a renderer will allow an upstream filter to change dynamically. However, at a very minimum, it should allow the upstream filter to change palettes. When an upstream filter changes media types, it will attach the format to the first sample delivered in that new type. If the renderer holds many samples in a queue waiting to be rendered, it should delay changing the format until the sample with the type change is actually about to be rendered.

Whenever a format change is detected by the video renderer, it should send an EC_DISPLAY_CHANGED notification. Most video renderers pick a format during connection so that the format can be drawn efficiently through GDI. If the user changes the current display mode without restarting the computer, a renderer might find itself with a bad image format connection and should send this notification. The first parameter should be the pin that needs reconnecting. The filter graph manager will arrange for the filter graph to be stopped and the pin reconnected. During the subsequent reconnection, the renderer can accept a more appropriate format.

Whenever a video renderer detects a palette change in the stream it should send the EC_PALETTE_CHANGED notification to the filter graph manager. The DirectShow video renderers detect whether a palette has really changed in dynamic format or not. The video renderers do this not only to filter out the number of **EC_PALETTE_CHANGED** notifications sent but also to reduce the amount of palette creation, installation, and deletion required.

Finally, the video renderer might also detect that the size of the video has changed, in which case, it should send the EC_VIDEO_SIZE_CHANGED notification. An application might use this notification to negotiate space in a compound document. The actual video dimensions are available through the IBasicVideo control interface. The DirectShow renderers detect whether the video has actually changed size or not prior to sending these events.

Handling Persistent Properties

All properties set through the IBasicVideo and IVideoWindow interfaces are meant to be persistent across connections. Therefore, disconnecting and reconnecting a renderer should show no effects on the window size, position, or styles. However, if the video dimensions change between connections, the renderer should reset the source and destination rectangles to their defaults. The source and destination positions are set through the **IBasicVideo** interface.

Both IBasicVideo and IVideoWindow provide enough access to properties to allow an application to save and restore all the data in the interface in a persistent format. This will be useful to applications that must save the exact configuration and properties of filter graphs during an editing session and restore them later.

Handling EC_REPAINT Notifications

The EC_REPAINT notification is sent only when the renderer is either paused or stopped. This notification signals to the filter graph manager that the renderer needs data. If the filter graph is stopped when it receives one of these notifications, it will pause the filter graph, wait for all filters to receive data (by calling GetState), and then stop it again. When stopped, a video renderer should hold on to the image so that subsequent WM_PAINT messages can be handled.

Therefore, if a video renderer receives a WM_PAINT message when stopped or paused, and it has nothing with which to paint its window, then it should send EC_REPAINT to the filter graph manager. If an **EC_REPAINT** notification is received while paused, then the filter graph manager calls IMediaPosition::put_CurrentPosition with the current position (that is, seeks to the current position). This causes the source filters to flush the filter graph and causes new data to be sent through the filter graph.

A renderer must send only one of these notifications at a time. Therefore, once the renderer sends a notification, it should ensure no more are sent until some samples are delivered. The conventional way to do this is to have a flag to signify that a repaint can be sent, which is turned off after an EC_REPAINT notification is sent. This flag should be reset once data is delivered or when the input pin is flushed, but not if end-of-stream is signaled on the input pin.

If the renderer does not monitor its EC_REPAINT notifications, it will flood the filter graph manager with **EC_REPAINT** requests (which are relatively expensive to process). For example, if a renderer has no image to draw, and another window is dragged across the window of the renderer in a full-drag operation, the renderer receives multiple WM_PAINT messages. Only the first of these should generate an **EC_REPAINT** event notification from the renderer to the filter graph manager.

A renderer should send its input pin as the first parameter to the EC_REPAINT notification. By doing this, the attached output pin will be queried for IMediaEventSink, and if supported, the **EC_REPAINT** notification will be sent there first. This allows output pins to handle repaints before the filter graph must be touched. This will not be done if the filter graph is stopped, because no buffers would be available from the decommitted renderer allocator.

If the output pin cannot handle the request and the filter graph is running, then the EC_REPAINT notification is ignored. An output pin must return NOERROR (S_OK) from IMediaEventSink::Notify to signal that it processed the repaint request successfully. The output pin will be called on the filter graph manager worker thread, which avoids having the renderer call the output pin directly, and so sidesteps any deadlock issues. If the filter graph is stopped or paused and the output doesn't handle the request, then the default processing is done.

Handling Notifications in Full-Screen Mode

The IVideoWindow plug-in distributor (PID) in the filter graph manages full-screen playback. It will swap a video renderer out for a specialist full-screen renderer, stretch a window of a renderer to full screen, or have the renderer implement full-screen playback directly. To interact in full-screen protocols, a video renderer should send an EC_ACTIVATE notification whenever its window is either activated or deactivated. In other words, an **EC_ACTIVATE** notification should be sent for each WM_ACTIVATEAPP message a renderer receives.

When a renderer is being used in full-screen mode, these notifications manage the switching into and out of that full-screen mode. Window deactivation typically occurs when a user presses ALT+TAB to switch to another window, which the DirectShow full-screen renderer uses as a cue to return to typical rendering mode.

When the EC_ACTIVATE notification is sent to the filter graph manager upon switching out of full-screen mode, the filter graph manager sends an EC_FULLSCREEN_LOST notification to the controlling application. The application might use this notification to restore the state of a full-screen button, for example. The **EC_ACTIVATE** notifications are used internally by DirectShow to manage full-screen switching on cues from the video renderers.

Summary of Notifications

This section lists the filter graph notifications that a renderer can send.

Event notification	Description
EC_ACTIVATE	Sent by video renderers in full-screen rendering mode for each WM_ACTIVATEAPP message received.
EC_COMPLETE	Sent by renderers after all data has been rendered.
EC_DISPLAY_CHANGED	Sent by video renderers when a display format changes.
EC_PALETTE_CHANGED	Sent whenever a video renderer detects a palette change in the stream.
EC_REPAINT	Sent by stopped or paused video renderers when a WM_PAINT message is received and there is no data to display. This causes the filter graph manager to generate a frame to paint to the display.
EC_USERABORT	Sent by video renderers to signal a closure that the user requested (for example, a user closing the video window).
EC_VIDEO_SIZE_CHANGED	Sent by video renderers whenever a change in native video size is detected.
EC_WINDOW_DESTROYED	Sent by video renderers when the filter is removed or destroyed so that resources that depend on window focus can be passed to other filters.

Exposing Capture and Compression Formats

This article describes how to return capture and compression formats by using the IAMStreamConfig::GetStreamCaps method. This method can get more information about accepted media types than the traditional way of enumerating a pin's media types, so it should typically be used instead. See Establishing Media Type Connections for information about traditional media type enumeration. **IAMStreamConfig::GetStreamCaps** can return information about the kinds of formats allowed for audio or video. Additionally, this article provides some sample code that demonstrates how to reconnect the input pin of a transform filter to ensure your filter can produce a particular output.

The IAMStreamConfig::GetStreamCaps method returns an array of pairs of media type and capabilities structures. The media type is an AM_MEDIA_TYPE structure and the capabilities are represented either by an AUDIO_STREAM_CONFIG_CAPS structure or a VIDEO_STREAM_CONFIG_CAPS structure. The first section in this article presents a video example and the second presents an audio example.

Contents of this article:

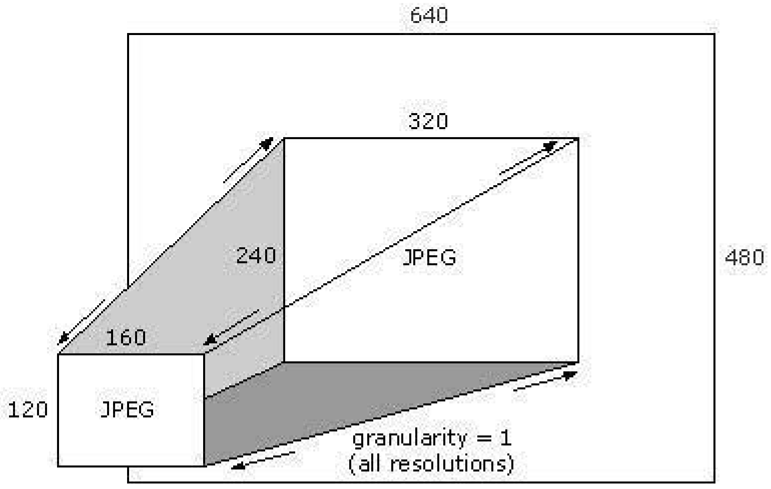
Video Capabilities

The IAMStreamConfig::GetStreamCaps method presents video capabilities in an array of pairs of AM_MEDIA_TYPE and VIDEO_STREAM_CONFIG_CAPS structures. You can use this to expose all the formats and resolutions supported on a pin as discussed below.

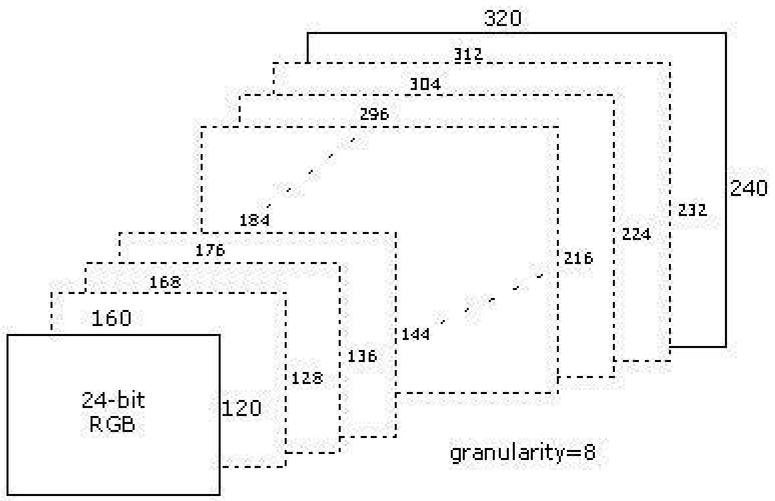
See Audio Capabilities for audio-related examples of IAMStreamConfig::GetStreamCaps.

Suppose your capture card supports JPEG format at all resolutions between 160 × 120 pixels and 320 × 240 pixels, inclusive. The difference between supported resolutions is one in this case because you add or subtract one pixel from each supported resolution to get the next supported resolution. This difference in supported resolutions is called granularity.

Suppose you card also supports the size 640 × 480. The following illustrates this single resolution and the above range of resolutions (all sizes between 160 × 120 pixels and 320 × 240 pixels).



Also, suppose it supports 24-bit color RGB format at resolutions between 160 × 120 and 320 × 240, but with a granularity of 8. The following illustration shows some of the valid sizes in this case.



To put it another way, and listing more resolutions, the following are all among the list of valid resolutions.

- 160 × 120
- 168 × 120
- 168 × 128
- 176 × 128
- 176 × 136
- ... additional resolutions ...
- 312 × 232
- 320 × 240

Use GetStreamCaps to expose these color format and dimension capabilities by offering a media type of 320 × 240 JPEG (if that is your default or preferred size) coupled with minimum capabilities of 160 × 120, maximum capabilities of 320 × 240, and a granularity of 1. The next pair you expose by using **GetStreamCaps** is a media type of 640 × 480 JPEG coupled with a minimum of 640 × 480 and a maximum of 640 × 480 and a granularity of 0. The third pair includes a media type of 320 × 240, 24-bit RGB with minimum capabilities of 160 × 120, maximum capabilities of 320 × 240, and a granularity of 8. In this way you can publish almost every format and capability your card might support. An application that must know what compression formats you provide can get all the pairs and make a list of all the unique subtypes of the media types.

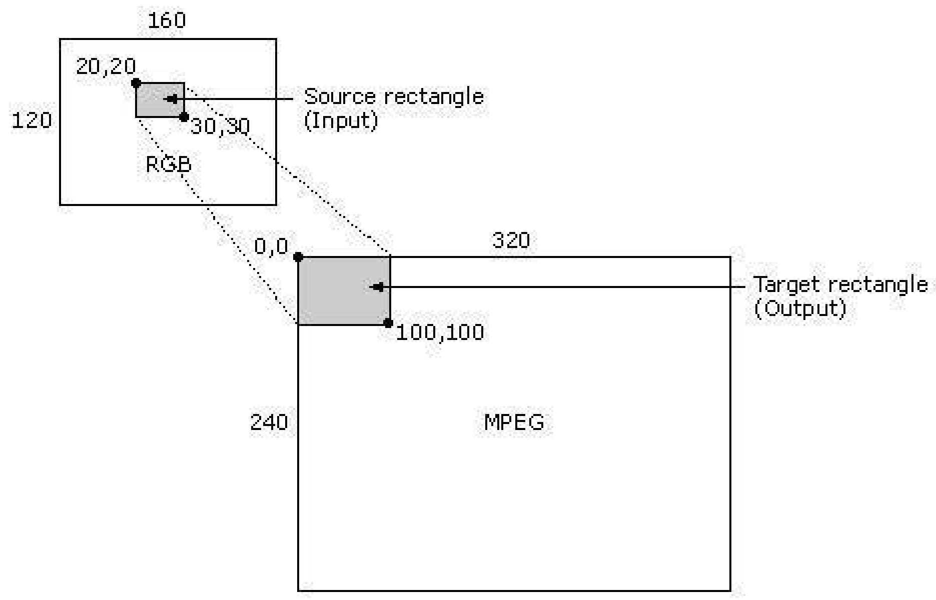
A filter obtains its media type source and target rectangles from the VIDEOINFOHEADER structure's rcSource and rcTarget members, respectively. Filters do not have to support source and target rectangles.

The cropping rectangle described throughout the IAMStreamConfig documentation is the same as the VIDEOINFOHEADER structure's rcSource rectangle for the output pin.

The output rectangle described throughout the IAMStreamConfig documentation is the same as the **biWidth** and **biHeight** members of the output pin's BITMAPINFOHEADER structure.

If a filter's output pin is connected to a media type with nonempty source and target rectangles, then your filter is required to stretch the input format's source subrectangle into the output format's target subrectangle. The source subrectangle is stored in the VIDEO_STREAM_CONFIG_CAPS structure's InputSize member.

For example, consider the following video compressor scenario: The input image is in RGB format and has a size of 160 × 120 pixels. The source rectangle's upper-left corner is at coordinate (20,20), and its lower-right corner is at (30,30). The output image is in MPEG format with a size of 320 × 240. The target rectangle's upper-left corner is at (0,0) and its lower-right corner is at (100,100). In this case, the filter should take a 10 × 10 piece of the 160 × 120 RGB source bitmap, and make it fill the top 100 × 100 area of a 320 × 240 bitmap, leaving the rest of the 320 × 240 bitmap untouched. The following illustration shows this scenario.



A filter might not support this and can fail to connect with a media type where rcSource and rcTarget are not empty.

The VIDEOINFOHEADER structure exposes information about a filter's data rate capabilities. For example, suppose you connected your output pin to the next filter with a certain media type (either directly or by using the media type passed by the CMediaType::SetFormat function). Look at the dwBitRate member of that media type's **VIDEOINFOHEADER** format structure to see what data rate you should compress the video to. If you multiply the number of units of time per frame in the **VIDEOINFOHEADER** structure's AvgTimePerFrame member by the data rate in the **dwBitRate** member and divide by 10,000,000 (the number of units per second), you can figure out how many bytes each frame should be. You can produce a smaller sized frame, but never a larger one. To determine the frame rate for a video compressor or for a capture filter, use **AvgTimePerFrame** from your output pin's media type.

Audio Capabilities

For audio capabilities, IAMStreamConfig::GetStreamCaps returns an array of pairs of AM_MEDIA_TYPE and AUDIO_STREAM_CONFIG_CAPS structures. As with video, you can use this to expose all kinds of audio capabilities on the pin, such as data rate and whether it supports mono or stereo.

See Video Capabilities for video-related examples relating to IAMStreamConfig::GetStreamCaps.

Suppose you support pulse code modulation (PCM) wave format (as represented by the Microsoft® Win32® PCMWAVEFORMAT structure) at sampling rates of 11,025, 22,050, and 44,100 samples per second, all at 8- or 16-bit mono or stereo. In this case, you would offer two pairs of structures. The first pair would have an AUDIO_STREAM_CONFIG_CAPS capability structure saying you support a minimum of 11,025 to a maximum of 22,050 samples per second with a granularity of 11,025 samples per second (granularity is the difference between supported values); an 8-bit minimum to a 16-bit maximum bits per sample with a granularity of 8 bits per sample; and one-channel minimum and two-channel maximum. The first pair's media type would be your default PCM format in that range, perhaps 22 kilohertz (kHz), 16-bit stereo. Your second pair would be a capability showing 44,100 for both minimum and maximum samples per second; 8-bit (minimum) and 16-bit (maximum) bits per sample, with a granularity of 8 bits per sample; and one-channel minimum and two-channel maximum. The media type would be your default 44 kHz format, perhaps 44 kHz 16-bit stereo.

If you support non-PCM wave formats, the media type returned by this method can show which non-PCM formats you support (with a default sample rate, bit rate, and channels) and the capabilities structure accompanying that media type can describe which other sample rates, bit rates, and channels you support.

Reconnecting Your Input to Ensure Specific Output Types

Filters implement the IAMStreamConfig::SetFormat method to set the audio or video stream's format before pins are connected. Additionally, if your output pin is already connected and you can provide a new type, then reconnect your pin. If the other pin your filter is connected to can't accept the media type, fail this call and leave your connection alone.

Transform filters that do not know what output types their pins can provide should refuse any calls to IAMStreamConfig::SetFormat and IAMStreamConfig::GetStreamCaps with the error code VFW_E_NOT_CONNECTED until their input pin is connected.

If your pin knows what types it can provide even when your input is not connected, it is okay to offer and accept them as usual. See Connecting Transform Filters for more information.

In certain cases it is useful to reconnect pins when you are offering a format on an established connection. For example, if you can compress video into format X but only if you get 24-bit RGB input, and you can turn 8-bit RGB input into compressed format Y, you can either:

1. Offer and accept both X and Y in IAMStreamConfig::GetStreamCaps and IAMStreamConfig::SetFormat all the time, or,
2. Only offer format X if your input is connected as 24, and only offer Y if your input is connected as 8. Fail both IAMStreamConfig::GetStreamCaps and IAMStreamConfig::SetFormat if your input is not connected.

No matter which one you choose, you will need some reconnecting code that looks like this:

```
// Overridden to do fancy reconnecting footwork.
//
HRESULT MyOutputPin::CheckMediaType(const CMediaType *pmtOut)
{
    HRESULT hr;
    CMediaType *pmtEnum;
    BOOL fFound = FALSE;
    IEnumMediaTypes *pEnum;

    if (!m_pFilter->m_pInput->IsConnected()) {
        return VFW_E_NOT_CONNECTED;
    }

    // Quickly verify that the media type is not bogus here
    //

    // If somebody has previously called SetFormat, fail this call if the media typ
    // isn't an exact match.

       // Accept this output type like normal; nothing fancy required.
    hr = m_pFilter->CheckTransform(&m_pFilter->m_pInput->CurrentMediaType(),
                                        pmtOut);
    if (hr == NOERROR)
        return hr;

    DbgLog((LOG_TRACE,3,TEXT("Can't accept this output media type")));
    DbgLog((LOG_TRACE,3,TEXT(" But how about reconnecting our input...")));

    // Attempt to find an acceptable type by reconnecting our input pin.
    // The pin our input pin connects to might be able to provide a type
    // that our pin can convert into the necessary type.
    hr = m_pFilter->m_pInput->GetConnected()->EnumMediaTypes(&pEnum);
    if (hr != NOERROR)
        return E_FAIL;
    while (1) {
        hr = pEnum->Next(1, (AM_MEDIA_TYPE **)&pmtEnum, &j);

        // All out of enumerated types.
        if (hr == S_FALSE || j == 0) {
            break;
        }

        // Can our pin convert between these?
        hr = m_pFilter->CheckTransform(pmtEnum, pmtOut);

        if (hr != NOERROR) {
            DeleteMediaType(pmtEnum);
            continue;
        }

        // OK, it offers an acceptable type, but will it accept it now?
        hr = m_pFilter->m_pInput->GetConnected()->QueryAccept(pmtEnum);
        // Nope.
        if (hr != NOERROR) {
            DeleteMediaType(pmtEnum);
            continue;
        }
        // OK, I'm satisfied.
        fFound = TRUE;
        DbgLog((LOG_TRACE,2,TEXT("This output type is only acceptable after reconn

        // All done with this.
        DeleteMediaType(pmtEnum);
        break;
    }
    pEnum->Release();

    if (!fFound)
        DbgLog((LOG_TRACE,3,TEXT("*NO! Reconnecting our input won't help")));

    return fFound ? NOERROR : VFW_E_INVALIDMEDIATYPE;
}


HRESULT MyOutputPin::SetFormat(AM_MEDIA_TYPE *pmt)
{
    HRESULT hr;
    LPWAVEFORMATEX lpwfx;
    DWORD dwSize;

    if (pmt == NULL)
        return E_POINTER;
    // To make sure streaming isn't in the middle of starting/stopping:
    CAutoLock cObjectLock(&m_pFilter->m_csFilter);

    if (m_pFilter->m_State != State_Stopped)
        return VFW_E_NOT_STOPPED;

    // Possible output formats depend on the input format.
    if (!m_pFilter->m_pInput->IsConnected())
        return VFW_E_NOT_CONNECTED;

    // Already using this format.
    if (IsConnected() && CurrentMediaType() == *pmt)
        return NOERROR;

    // See if this type is acceptable.
    if ((hr = CheckMediaType((CMediaType *)pmt)) != NOERROR) {
        DbgLog((LOG_TRACE,2,TEXT("IAMStreamConfig::SetFormat rejected")));
        return hr;
    }

    // If connecting to another filter, make sure they like it.
    if (IsConnected()) {
        hr = GetConnected()->QueryAccept(pmt);
        if (hr != NOERROR)
            return VFW_E_INVALIDMEDIATYPE;
    }

    // Now make a note that from now on, this is the only format allowed,
    // and refuse anything but this in the CheckMediaType code above.

    // Changing the format means reconnecting if necessary.
    if (IsConnected())
        m_pFilter->m_pGraph->Reconnect(this);

    return NOERROR;
}


// Overridden to complete our fancy reconnection footwork:
//
HRESULT MyWrapper::SetMediaType(PIN_DIRECTION direction,const CMediaType *pmt)
{
    HRESULT hr;

    // Set the OUTPUT type.
    if (direction == PINDIR_OUTPUT) {

        // Uh oh.  As part of our fancy reconnection, our input pin might be asked
        // provide a media type it cannot provide without reconnection
        // to a different type.
        if (m_pInput && m_pInput->IsConnected()) {

            // If our pin can actually provide this type now, don't worry.
            hr = CheckTransform(&m_pInput->CurrentMediaType(),
                                &m_pOutput->CurrentMediaType());
            if (hr == NOERROR)
                return hr;

                DbgLog((LOG_TRACE,2,TEXT("*Set OUTPUT requires RECONNECT of INPUT!

            // Reconnect our input pin.
            return m_pGraph->Reconnect(m_pInput);

        }
        return NOERROR;
    }

    return NOERROR;
}
```

DirectShow COM Interfaces

This section contains reference entries for all the DirectShow COM interfaces and their methods.

Previous Home Topic Contents Index Next

Summary of DirectShow COM Interfaces

This article groups the Microsoft® DirectShow™ interfaces according to the objects that expose them. It explains which object implements each interface and who is likely to call the interface methods implemented on each type of object. The information is presented as a series of tables of COM interfaces relating to each object, and so provides a summary and quick reference for understanding the DirectShow interfaces.

Contents of this article:

Introducing the DirectShow COM Interfaces

The DirectShow COM interfaces comprise the schematic of an architecture for streaming time-stamped media. The filter graph, through which media flows, is composed of objects, such as filters, pins, media samples, allocators, and enumerators, that work together. COM interfaces are implemented on these objects and are called by other objects with which they interact. The filter is the only filter graph COM object that has a CLSID; all other objects in the filter graph support COM interfaces, and are created as needed by the filter. Filters and their supporting object must implement their COM interfaces and a class library is available for help in that task. The filter graph manager, on the other hand, has a CLSID and supports several fully implemented interfaces, as do plug-in distributors, which are aggregated by the filter graph manager. (Microsoft-provided plug-in distributors are referred to as the filter graph manager in this article.)

This section contains the following topics.

The DirectShow COM interfaces can be categorized as follows:

- Filter graph manager interfaces, which are fully implemented and used by applications to create, connect, and control filter graphs and by filters within the filter graph to post event notifications and to force reconnections when needed.

- Filter and pin interfaces, which must be implemented by the filter. They comprise the methods exposed by filters for communicating with the filter graph manager, connecting with other filters, passing data downstream (from source filter to renderer filter) and passing quality control and media positioning information upstream (from renderer to source).
- Enumerator and media sample interfaces, which are interfaces on objects created temporarily for passing information.
- Control interfaces, which are exposed by filters and the filter graph manager to enable the starting, stopping, and positioning of media in the stream. The control interfaces on the filters must be implemented when writing a filter, whereas they are already implemented on the filter graph manager.

Filter Graph Manager Interfaces

Most filter graph manager interfaces fall in the category of control interfaces, also listed separately in this article; however, some are unique to the filter graph manager.

Interface	**What calls methods**	**What methods do**	**Comments**
IAMCollection	An Automation client such as Microsoft® Visual Basic®.	Retrieve the number of items in the collection and retrieve an indexed item.	Implemented by the filter graph manager. Not used by C or C++ applications.
IDeferredCommand	The application that sent a deferred command using IQueueCommand.	Retrieve confidence information, postpone or cancel a deferred command, and return HRESULT values from an invoked command.	Implemented by the filter graph manager and returned to any application that calls IQueueCommand methods.
IEnumFilters	Application or possibly a filter that needs to know what other filters are in the graph.	Retrieve filters, skip filters, or clone the enumerator.	Implemented in the filter graph manager.
IEnumRegFilters	Filter mapper.	Retrieve filters, skip items, or clone the enumerator.	Implemented by the filter graph manager.
IFilterGraph	Not usually called directly.	Add, connect, and reconnect filters in a filter graph.	Use IGraphBuilder instead, because it inherits from this interface.
IFilterGraph2	C or C++ applications.	Extend filter graph functionality.	
IFilterInfo	An Automation client such as Visual Basic.	Retrieve name, vendor information, IBaseFilter interface, file name (for source filters), specified pin object, or collection of associated pin objects.	Implemented by the filter graph manager. Not used by C or C++ applications.
IGraphBuilder	Application.	Create filter graphs dynamically from stream media type or re-create stored filter graph.	Inherits from IFilterGraph. Also uses a filter mapper object to look up filters in the registry.

IGraphVersion	Application and plug-in distributors in the filter graph manager.	Retrieve the current filter graph version and determine when a filter graph has had filters added, deleted, or reconnected.	
IMediaControl	Application.	Run, pause, and stop the filter graph, and retrieve the state.	Has many of the same methods as the IMediaFilter interface, which is implemented on filters.
IMediaEvent	Applications that need to retrieve events passed to the filter graph manager from filters.	Get events, get event handles, block until completion, and block or unblock default handling of events by the filter graph manager.	Implemented by the filter graph manager.
IMediaEventSink	Filters that need to pass events to the application.	Receive event notifications from filters.	Implemented by the filter graph manager.
IMediaPosition	Applications.	Get duration and position properties, and get and set start time, stop time, preroll time, and rate properties.	Implemented on the filter graph manager and also on filters.
IMediaSeeking	Applications.	Set and retrieve current position and stop position in units other than time (such as sample or field).	Implemented on the filter graph manager and also on filters.
IMediaTypeInfo	An Automation client such as Visual Basic.	Retrieve the major and minor media types.	Implemented by the filter graph manager. Not used by C or C++ applications.
IPinInfo	An Automation client such as Visual Basic.	Retrieve pin information such as name, direction, connections, and collection of associated media type objects. Also includes methods to connect and disconnect pins.	Implemented by the filter graph manager. Not used by C or C++ applications.
IQueueCommand	Application needing to send a deferred command.	Cue commands to run at stream time (offset from start) or presentation time.	Implemented by the filter graph manager and used by applications. Filters can implement this.
IRegFilterInfo	An Automation client such as Visual Basic.	Retrieve a filter name and add it to the filter graph.	Implemented by the filter graph manager. Not used by C or C++ applications.
ISeekingPassThru	Applications.	Instantiate and initialize a CRendererPosPassThru object. You can use this object to keep track of reference times and stream times.	Implemented on video renderer filters that need to keep track of reference time and stream time.

Filter and Pin Interfaces

Filters are composed of one filter object and one or more pin objects. Although only the IUnknown and IBaseFilter interfaces are strictly required on a filter, filters can support other filter and pin interfaces, as discussed in Filter Interfaces.

This topic contains the following subtopics.

- Filter Interfaces
- Pin Interfaces

Filter Interfaces

The following interfaces are exposed by filter objects in order to be integrated with the filter graph manager. The filter is the main COM object and has a class ID (CLSID) and name registered in the registry. Filters must provide access to their pins and otherwise communicate with the filter graph. They must also allow the filter graph manager to manage the data flow by accepting state change messages.

Interface	**What calls methods**	**What methods do**	**Comments**
IAMovieSetup	Entry-point routines in Dllentry.cpp.	Register and unregister the object.	This is implemented by the base classes for most of what is required to make a filter self-registerable. Need to override one base member function to provide setup structures.
IBaseFilter	Filter graph manager.	Same as IMediaFilter plus enumerate pins, retrieve filter and vendor information, and locate pins when rebuilding a persistent filter graph.	Inherits methods from the IMediaFilter interface. Implemented by the CBaseFilter class.
IMediaFilter	Nothing directly.	Put the filter in run, stop, or pause state, get and set the reference clock, and retrieve the filter state.	Inherited by the IBaseFilter interface, which should be used instead of referencing this directly.
IPersist	Filter graph manager when loading preconfigured filter graph files.	Retrieve the filter's class identifier.	Inherited by IBaseFilter along with IMediaFilter.
IUnknown	Filter graph manager.	Retrieve a pointer to the interface, add and delete references to the interface.	Implemented by the CUnknown base class.

The following additional filter interfaces can be exposed by filters such as source filters, and by video and audio renderers. Source filters are notified by means of a quality control mechanism so that they can adjust the amount of data introduced to the stream according to the renderer's performance. Audio renderers are usually called upon to provide a reference clock, since audio hardware generates this. The video renderer filter supplied by Microsoft exposes interfaces to handle both the video window and the transferring of video frames into video buffers.

Interface	**What calls methods**	**What methods do**	**Comments**
IAMCrossbar	C or C++ applications.	Route messages from an analog or digital audio or video source to a video capture filter.	Exposed on analog video crossbar filters.
IAMExtDevice	C or C++ applications.	Control external devices.	
IAMExtTransport	C or C++ applications.	Control specific behaviors of an external VCR.	
IAMTimecodeDisplay	C or C++ applications.	Define behavior of an external SMPTE/MIDI timecode display device.	
IAMTimecodeGenerator	C or C++ applications.	Specify how an external SMPTE/MIDI timecode generator should supply data to the filter graph, and the formats in which timecode should be supplied.	
IAMTimecodeReader	C or C++ applications.	Specify the timecode format that an external device should read and how it is embedded in the media.	
IAMTVTuner	C or C++ applications.	Enables applications to set TV transmission types.	
IAMVideoProcAmp	C or C++ applications.	Control video quality settings.	Exposed on WDM video capture filters.
IAsyncReader	Downstream parser filter.	Perform synchronized reads, request data, request allocator, begin and end flushing, and retrieve file's total length.	Implemented on the Async Sample (Asynchronous Reader Filter), which reads media types with a major type of MEDIATYPE_Stream.
IBasicAudio	Applications and Automation clients such as Visual Basic.	Get and set the properties of the audio renderer filter.	Can be implemented on an audio renderer filter. Supports Automation.
IBasicVideo	Applications and Automation clients such as Visual Basic.	Get and set the source video rectangle, and retrieve video size, palette values, and the current image.	Usually implemented on a video renderer filter. Supports Automation.

ICreateDevEnum	C or C++ applications.	Enumerate hardware devices.	
IDirectDrawVideo	C or C++ applications.	Set and retrieve DirectDraw® hardware and emulated capabilities and surface types.	Usually implemented on a video renderer filter.
IFileSinkFilter	Any application that needs to set the name of the file from which the file source filter will read.	Set or retrieve the file name.	Implemented on a file writer filter, as used in a video capture filter graph.
IFileSinkFilter2	C or C++ applications.	Set or retrieve the file name, optionally overwriting an existing file.	
IFileSourceFilter	Any application that needs to set the name of the file into which the file sink filter will write.	Set or retrieve the file name.	Implemented on any source filter that needs a file name from the user.
IFullScreenVideo	C or C++ applications.	Set and retrieve full-screen modes, message drain, icon caption, and other information.	Usually implemented on a video renderer filter.
IMediaPropertyBag	C or C++ applications.	Expose copyright information on filters.	
IMediaSample2	Filters.	Expose sample properties.	
IOverlay	Filters upstream from the renderer that need to be notified of window changes.	Set and retrieve palette and color key information; get window handle, clip list, window position; set up advise link with upstream filter.	Usually implemented on a video renderer filter.
IQualityControl	Filter graph manager, upstream filter, or pin.	Receive a quality message and receive the quality sink location.	Normally implemented on filters that can affect the quality when they receive the message.
IQualProp	Property page objects.	Retrieve rendering quality properties of the video renderer, such as the number of frames drawn, jitter, and so on.	Supports Automation.

IReferenceClock	Filters that need to be synchronous with a reference clock.	Register for time notifications from the filter, convert real to reference time, and retrieve the current time.	Implemented on a filter that can generate a reference clock, typically an audio renderer. Provides services similar to the timeBeginPeriod and timeSetEvent Win32® functions.
IVideoWindow	Applications and Automation clients such as Visual Basic.	Control the window aspects of a video renderer.	Usually implemented on a video renderer filter.

DirectShow provides filters that implement particular interfaces for you. Applications typically use those interfaces, but filters do so as well.

Interface	**What calls methods**	**What methods do**	**Comments**
IAMDirectSound	DirectSound audio renderer.	Set and retrieve the window that will handle the sound playback.	DirectSound audio renderer implements and uses this interface.
IAMLine21Decoder	Applications or video mixer filter.	Provide access to closed caption settings.	Line21 decoder implements this interface.
IAMStreamSelect	Applications.	Control which logical streams are played and retrieve information about them.	The MPEG splitter implements this interface.
IAMVfwCaptureDialogs	Applications.	Provide access to dialog boxes exposed by Video for Windows capture drivers.	Video for Windows capture filter implements this interface.
IAMVfwCompressDialogs	Applications.	Provide access to dialog boxes exposed by Video for Windows compressors.	Video for Windows installable compression manager (ICM) filter implements this interface.
IAMAudioCutListElement	Applications and filters.	Provide support for a cutlist element for an audio file stream in a WAV or AVI file.	The CLSID_AudioFileClip object implements this interface.
ICaptureGraphBuilder	Applications.	Simplify building capture filter graphs.	Capture graph builder object implements this interface.
IConfigAviMux	Applications.	Control how the AVI multiplexer filter writes files to disk.	AVI multiplexer filter implements this interface on its property page.
IConfigInterleaving	Applications.	Control how the AVI multiplexer filter writes files to disk and set interleaving configuration information.	AVI multiplexer filter implements this interface on its property page.

IAMCutListElement	Filters.	Describe a base object, which represents an element in a cutlist.	DirectShow provides the CLSID_VideoFileClip and CLSID_AudioFileClip objects, which can create an object that implements it for you.
ICutListGraphBuilder	Applications.	Enable you to easily implement one or more cutlist filter graphs.	
IDvdControl	Applications.	Control playback and searching on DVD discs.	DVD navigator filter implements this interface.
IDvdGraphBuilder	Applications.	Simplify building DVD filter graphs.	DVD graph builder object implements this interface.
IDvdInfo	Applications.	Query for DVD attributes and DVD player status.	DVD navigator filter implements this interface.
IFileClip	Applications.	Provide a simple way for an application to create one or more cuts from a single media file, or to create blank cuts.	
IAMFileCutListElement	Filters.	Provide support for a cutlist element for a file stream.	DirectShow provides the CLSID_VideoFileClip and CLSID_AudioFileClip objects that implement it for you.
IStandardCutList	Applications.	Provide a simple way for an application to feed a cutlist into a cutlist provider (filter).	
IAMVideoCutListElement	Filters.	Provide support for a cutlist element from an AVI video file stream.	DirectShow provides the CLSID_VideoFileClip object that implements it for you.
IVPBaseConfig	Video port mixer filter.	Enable a video port (VP) mixer filter to communicate with a VP driver.	Ksproxy filter implements this interface.
IVPBaseNotify	Applications.	Control properties of a filter that uses a video port.	Video port mixer filter implements this interface.
IVPConfig	Video port mixer filter.	Enable a video port (VP) mixer filter to communicate with a VP driver.	Ksproxy filter implements this interface.
IVPNotify	Applications.	Control properties of a filter that uses a video port.	Video port mixer filter implements this interface.

Pin Interfaces

Pin objects expose these interfaces. Pins do not usually have registered class identifiers and are usually created by the filter object on which they reside. They are exposed externally by the filter, which includes a method (IBaseFilter::EnumPins) to hand out pointers to the IPin interfaces of its pins, usually to the filter graph manager. The filter graph manager is responsible for connecting pins by calling an **IPin** method on one of the pins with a pointer to the other pin. Once pins are connected, each pin holds a pointer to the pin to which it is connected.

Interface	**What calls methods**	**What methods do**	**Comments**
IAMAudioInputMixer	Applications.	Adjust audio input characteristics.	Input pin of an audio capture filter typically implements this interface.
IAMBufferNegotiation	Applications.	Set and retrieve buffer properties.	The IAMBufferNegotiation interface informs a pin what kind of buffer specifications it should use when connected.
IAMDevMemoryAllocator	Applications.	Provide creation of third-party memory allocators.	Makes use of on-board memory manager objects.
IAMDevMemoryControl	Applications.	Control and identify on-board codec memory.	This interface is supported by a device memory control object.
IAMDroppedFrames	Applications.	Provide information about the number of dropped frames, frame rate, and data rate.	Capture filter's video output pin should implement this interface.
IAMStreamConfig	Applications or filters.	Provide types of formats an output pin can connect with.	Output pins of capture and compression filters typically implement this interface.
IAMStreamControl	Applications.	Enable control of streams in a filter graph.	Implemented by any input or output pins.
IAMVideoCompression	Applications.	Control compression parameters that aren't part of the media type.	Output pin of a video capture or compression filter typically implements this interface.
IKsPropertySet	Applications or filters.	Sets and retrieves device properties.	Expose device properties and enable an application or filter to change the properties.
IMediaPosition	Filter graph manager or downstream filter.	Get duration and position properties, and get and set start time, stop time, preroll time, and rate properties.	Downstream filters call methods on output pins supporting this to pass a requested media position upstream. Implemented in CPosPassThru on pins.
IMediaSeeking	Applications.	Set and retrieve current position and stop position in units other than time (such as sample or field).	Downstream filters call methods on output pins supporting this to pass a requested media position upstream. Implemented in CPosPassThru on pins.

IMemAllocator	Owning filter and output pin of connected filter.	Allocate one or more buffers based on required size, retrieve a buffer for a media sample, commit memory when in use, and release it (decommit) when not in use.	Appears on allocator object usually created by IMemInputPin. Implemented by the CMemAllocator class.
IMemInputPin	Filter graph manager, output pin of a connected filter.	Retrieve a preferred allocator, receive the allocator provided by output pin, receive media samples, and tell whether the pin will block on receive.	Usually only on input pins. Implemented by the CBaseInputPin base class.
IPin	Filter graph manager, other pins, the owning filter.	Connect and disconnect the pin, retrieve information on external and internal pin connections, retrieve preferred media types enumerator, negotiate preferred media types, receive flush and end-of-stream notifications.	Implemented on all pins by the CBasePin base class.
IQualityControl	Downstream filter or pin on downstream filter.	Receive a quality message and receive the quality sink to send quality messages to.	Implemented on output pins by the CBaseOutputPin base class, where it is used to pass the message upstream.
IUnknown	Filter graph manager, other pins, the owning filter.	Retrieve a pointer to the interface, add and delete references to the interface.	Implemented on all pins by the CUnknown base class.

In addition, the Microsoft video renderer's input pin supports the IOverlay interface, which allows the connected upstream pin to effectively register its IOverlayNotify interface in order to receive notifications of video window changes. Replacement video renderers can also implement this if they are intended to connect to the same filters as the video renderer provided with DirectShow.

Media Sample and Enumerator Interfaces

Media sample and enumerator interfaces are temporary objects created to pass information or data between objects. They do not have class identifiers.

This topic contains the following subtopics.

- Media Sample Interfaces
- Enumerator Interfaces

Media Sample Interfaces

The media sample interface, IMediaSample, is created from the memory allocator, which uses the media sample object as its unit of exchange. It has no class identifier. It is the unit of media data that is passed from one filter to the next via the memory allocator shared by two connected pins.

Interface	What calls methods	What methods do	Comments
IMediaSample	Pins or filters that need to manipulate the media sample data or examine its properties.	Retrieve a pointer to data, and get and set properties on the media sample such as buffer size, time stamp, data length, type, synchronization point, preroll, and end-of-stream properties.	Implemented on media samples by the CMediaSample base class.
IUnknown	Pin or filter.	Retrieve pointer to the interface, add and delete references to the interface.	Implemented on media samples by the CUnknown base class.

Enumerator Interfaces

Enumerators in DirectShow are based on the COM IEnumXXXX interfaces. They include the **Next** and **Prev** methods, which tell the enumerator what item or items to return; the **Skip** method, which skips one or more items; and the **Clone** method, which makes a copy of the enumerator. Enumerators are used to present lists of items such as filters in a filter graph, pins on a filter, or media types that are preferred by a pin.

Interface	What calls methods	What methods do	Comments
IEnumFilters	Application or possibly a filter that needs to know what other filters are in the graph.	Retrieve filters, skip filters, or clone the enumerator.	Implemented in the filter graph manager.
IEnumMediaTypes	Filter graph manager or connected pin negotiating a media type.	Retrieve media types, skip media types, or clone the enumerator.	Implemented by the CEnumMediaTypes class.
IEnumPins	Filter graph manager.	Retrieve pins, skip pins, or clone the enumerator.	Implemented by the CEnumPins class.
IEnumRegFilters	Filter mapper.	Retrieve filters, skip items, or clone the enumerator.	Implemented by the filter graph manager.

Control Interfaces

Control interfaces allow the filter graph manager to coordinate the activities of the data stream with filters. Interfaces described previously in both the Filter Graph Manager Interfaces and Filter and Pin Interfaces sections are repeated here so that all control interfaces can be viewed together.

Interface	What calls methods	What methods do	Comments
IBaseFilter	Filter graph manager.	Same as IMediaFilter plus enumerate pins, retrieve filter and vendor information, and locate pins when rebuilding a persistent filter graph.	Inherits methods from the IMediaFilter interface. Implemented by the CBaseFilter class.

IMediaControl	Application.	Run, pause, and stop the filter graph, and retrieve the state.	Implemented by the filter graph manager.
IMediaFilter	Nothing directly.	Put the filter in run, stop, or pause state, get and set the reference clock and retrieve the filter state.	Inherited by IBaseFilter, which should be used instead of referencing this directly.
IMediaPosition	Application, when exposed on the filter graph manager; filter graph manager or downstream filter, when exposed on a filter.	Get duration and position properties, and get and set start time, stop time, preroll time, and rate properties.	Implemented on the filter graph manager and on filters.
IMediaSeeking	Applications.	Set and retrieve current position and stop position in units other than time (such as sample or field).	Implemented on the filter graph manager and on filters.

Because DirectShow objects are COM-based objects, it is a natural extension to use other COM interfaces published in the COM specifications to perform functions such as listing property pages or accessing files. Following are some of the COM interfaces that are commonly used in DirectShow filters.

- IPersistFile
- ISpecifyPropertyPages

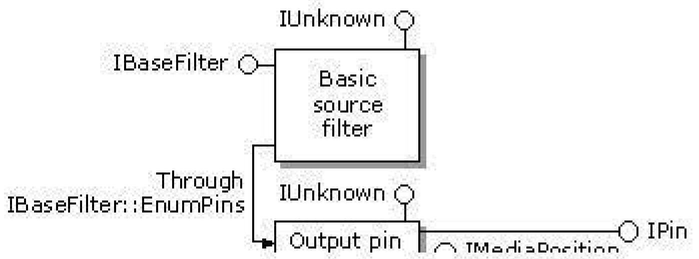
Interfaces on a Typical Filter Graph

Perhaps the best way to put the DirectShow interfaces into perspective is to look at the interfaces exposed in a simple filter graph. The example chosen here is a filter graph that plays back audio data stored on a disk. It is composed of a source filter and an audio renderer filter (no transforms are done, so this is a very simple graph).

The basic source filter consists of an IBaseFilter interface, and one output pin that can be found by using the IBaseFilter::EnumPins method. The COM IPersist interface might also be present on the filter (not shown here) to enable the filter to be stored as part of a persistent filter graph.

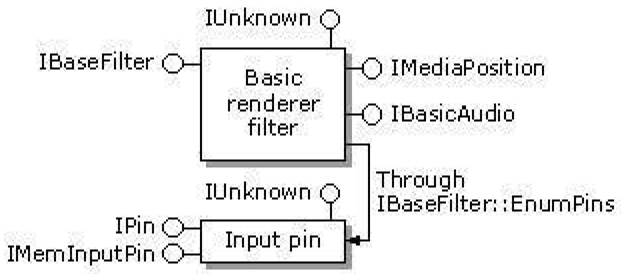
The output pin supports IPin and IMediaPosition, since the source filter is a seekable filter (that is, it can be told to move to a particular position in the media stream). The following illustration shows the interfaces on the source filter and its output pin.

Note that quality management is not implemented in this example, but could be by including the IQualityControl interface on the output pin as well.

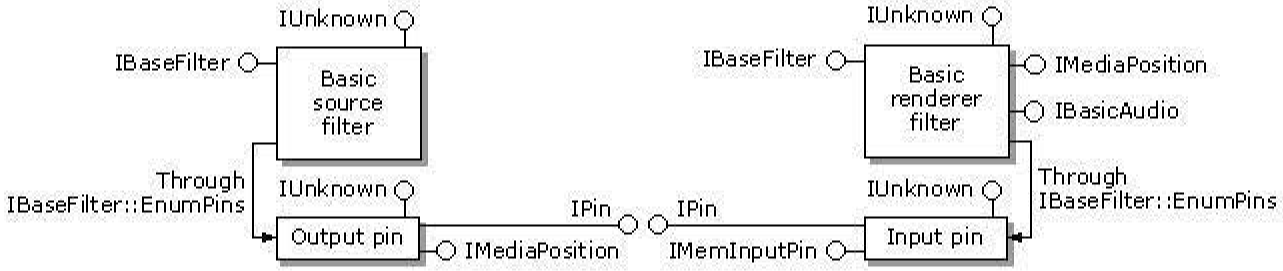


The basic renderer filter consists of a COM driver supporting the IBaseFilter (and, again, IMediaFilter by inheritance) and IMediaPosition interfaces, as well as having one input pin through the EnumPins method of the **IBaseFilter** interface. The COM IPersistFile interface might also be present on the filter (not shown here) in order load the file. The renderer can support additional interfaces, such as IReferenceClock (as a master for synchronization) or IBasicAudio, for an audio renderer as shown here.

The input pin supports the IPin and IMemInputPin interfaces. Methods of the IMediaPosition interface on the renderer filter can call the **IMediaPosition** interface methods of the output pin that is connected to the renderer's input pin.

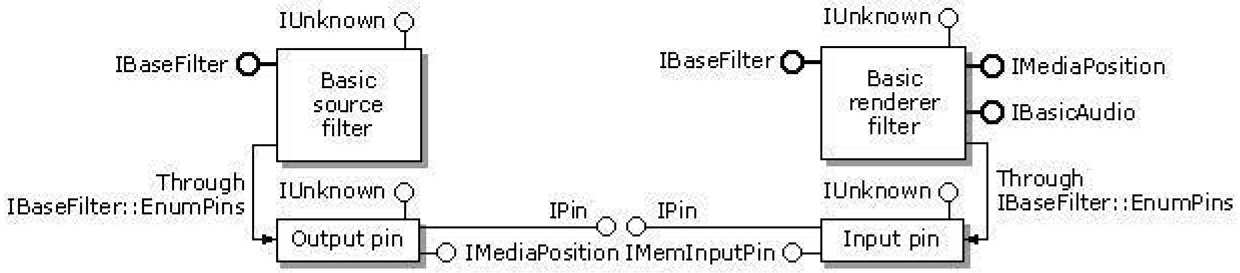


A basic representation of the connection of a source filter to a renderer filter follows:



Once the pins are enumerated and connected, it is the implementation of the output and input pins that defines the interactions of the two pins.

Once the filters are connected, the control available to the user (filter graph manager) is indicated through the bold interfaces. This continues with the example of connecting an audio source to an audio renderer.



The following interfaces are those that would likely be used during a transport.

- IMediaPosition supports the ability to seek to a position and change the rate.
- IBasicAudio supports the ability to set the volume.

- IBaseFilter supports such methods as Run and Pause (methods on the filter graph manager's IMediaControl interface).

Special (custom) interfaces that the filters would support must be obtained directly from the filters. Usage of these interfaces implies that the user is aware of their identifiers.

DirectShow Interfaces by Category

The following lists group the Microsoft® DirectShow™ interfaces according to whether application or filter developers typically call them, or whether they are exposed at the filter graph level. Both application and filter developers can call some interfaces, while others have a typical use by one or the other, but both types of developers can still call them. See the documentation about each interface for more information. Each of the categories (Application-Level Interfaces, Graph-Level Interfaces, and Filter-Level Interfaces) is divided into functional groupings.

In addition, the Multimedia Streaming Reference contains documentation on interfaces specific to multimedia streaming.

Application-Level Interfaces

- IAMCollection
- IAMCopyCaptureFileProgress
- IAMLine21Decoder
- IAMVfwCaptureDialogs
- IAMVfwCompressDialogs
- ICaptureGraphBuilder
- IDistributorNotify
- IEnumPins
- IFileSourceFilter
- IGraphBuilder
- IMediaStream
- IMultiMediaStream
- ISeekingPassThru
- IStreamSample
- IVPBaseNotify
- IVPNotify

Digital Versatile Disc (DVD) Application-Level Interfaces

- IDvdControl

- IDvdGraphBuilder
- IDvdInfo

Cutlist Application-Level Interfaces

- ICutListGraphBuilder
- IFileClip
- IStandardCutList

Cutlist Application-Level or Filter-Level Interfaces

- IAMAudioCutListElement
- IAMCutListElement
- IAMFileCutListElement
- IAMVideoCutListElement

Capture, Compression, Device Enumeration, and Windows Driver Model (WDM) Capture Application-Level or Filter-Level Interfaces

- IAMAudioInputMixer
- IAMBufferNegotiation
- IAMCrossbar
- IAMTVTuner
- IAMDroppedFrames
- IAMStreamConfig
- IAMStreamControl
- IAMStreamSelect
- IAMVideoCompression
- IAMVideoProcAmp
- IConfigAviMux
- IConfigInterleaving
- ICreateDevEnum
- IFileSinkFilter
- IFileSinkFilter2
- IMediaPropertyBag
- IMixerPinConfig

Device Control, Timecode, and Property Set Application-Level or Filter-Level Interfaces

- IAMExtDevice
- IAMExtTransport
- IAMTimecodeDisplay
- IAMTimecodeGenerator
- IAMTimecodeReader
- IKsPropertySet

Graph-Level Interfaces

- IAMovie

- IBasicAudio
- IBasicVideo
- IDeferredCommand
- IDirectDrawVideo
- IDistributorNotify
- IEnumFilters
- IEnumRegFilters
- IFilterGraph
- IFilterGraph2
- IFilterInfo
- IFilterMapper
- IFullScreenVideo
- IGraphVersion
- IMediaControl
- IMediaEvent
- IMediaEventEx
- IMediaEventSink
- IMediaPosition
- IMediaSeeking
- IMediaTypeInfo
- IQualityControl
- IQualProp
- IQueueCommand
- IReferenceClock
- IRegFilterInfo
- IResourceConsumer
- IResourceManager
- IVideoWindow

Filter-Level Interfaces

- IAMovieSetup
- IAsyncReader
- IBaseFilter
- IBasicAudio
- IBasicVideo
- IEnumMediaTypes
- IMediaFilter
- IMemInputPin
- IOverlay
- IOverlayNotify
- IPin
- IPinInfo
- IVPBaseConfig
- IVPConfig

DirectSound Filter-Level Interface

- IAMDirectSound

Memory Allocation and Media Sample Filter-Level Interfaces

- IAMDevMemoryAllocator
- IAMDevMemoryControl
- IMediaSample
- IMediaSample2
- IMemAllocator

COM Interfaces

- IUnknown

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

IAMAudioCutListElement Interface

The **IAMAudioCutListElement** interface provides support for a cutlist element for an audio file stream in a WAV or AVI file.

See About Cutlists and Using Cutlists for more information.

When to Implement

Usually, you don't need to implement this interface because DirectShow provides the CLSID_AudioFileClip object that implements it for you. Implement this interface in your application when you need to change the default behavior of this interface.

When to Use

Use this interface in your filter when you specify an audio-based media clip. Call QueryInterface on the IAMCutListElement interface to determine if the element is an audio type element.

When compiling a cutlist application you must explicitly include the cutlist header file as follows:

```
#include <cutlist.h>
```

Methods in Vtable Order

IUnknown methods	Description
QueryInterface	Retrieves pointers to supported interfaces.
AddRef	Increments the reference count.
Release	Decrements the reference count.

IAMAudioCutListElement methods	Description
GetStreamIndex	Retrieves the index to the stream in the AVI file.
HasFadeIn	Determines if the element fades in automatically.
HasFadeOut	Determines if the element fades out automatically.

IAMAudioCutListElement::GetStreamIndex

IAMAudioCutListElement Interface

Retrieves the index to the stream in the AVI file.

```
HRESULT GetStreamIndex(
  DWORD *piStream
  );
```

Parameters

piStream
[out] Stream number to be opened.

Return Values

Returns an HRESULT value that depends on the implementation of the interface. **HRESULT** can include one of the following standard constants, or other values not listed.

Value	Meaning
E_FAIL	Failure.
E_INVALIDARG	Argument is invalid.
E_NOTIMPL	Method is not supported.
E_POINTER	Null pointer argument.
S_OK	Success.

Remarks

This method must always retrieve zero for the stream index. For AVI files, only the first audio stream is supported.

IAMAudioCutListElement::HasFadeIn

IAMAudioCutListElement Interface

Determines if the element fades in automatically.

HRESULT HasFadeIn(void);

Return Values

Returns S_OK if the element should be automatically faded in, or S_FALSE if not.

Remarks

This method always returns S_OK, but fading in and out is not currently supported.

IAMAudioCutListElement::HasFadeOut

IAMAudioCutListElement Interface

Determines if the element fades out automatically.

HRESULT HasFadeOut(void);

Return Values

Returns S_OK if the element should be automatically faded out, or S_FALSE if not.

Remarks

This method always returns S_OK, but fading in and out is not currently supported.

IAMAudioInputMixer Interface

The **IAMAudioInputMixer** interface tells an audio capture filter what level, panning, and equalizer to use for each input. The name of each pin, such as "Line input 1" or "Mic", reflects the type of input.

Implementation of the methods on this interface depends on the device. A device might not implement all methods depending on its capabilities.

When to Implement

Implement this interface on each input pin of an audio capture filter. You can also implement this interface on the audio capture filter itself to control the overall record level and panning after the audio mixing occurs.

When to Use

Use this interface when your application needs to adjust audio input characteristics such as mixing of a particular input, use of mono or stereo, mix level, pan level, loudness, treble, and bass settings. Use the pin names to decide how to set the recording levels for each type of input.

Methods in Vtable Order

IUnknown methods	Description
QueryInterface	Retrieves pointers to supported interfaces.
AddRef	Increments the reference count.
Release	Decrements the reference count.

IAMAudioInputMixer methods	Description
put_Enable	Enables or disables an input in the mix.
get_Enable	Retrieves whether the input is enabled.
put_Mono	Combines all channels of an input into a mono signal.
get_Mono	Retrieves whether all channels of an input are combined into a mono signal.
put_MixLevel	Sets the record level for this input.
get_MixLevel	Retrieves the recording level for this input.
put_Pan	Sets the pan for this input.
get_Pan	Retrieves the pan for this input.
put_Loudness	Turns the loudness control for this input on or off.
get_Loudness	Retrieves the loudness control setting for this input.
put_Treble	Sets the treble equalization for this input.

get_Treble	Retrieves the treble equalization for this input.
get_TrebleRange	Retrieves the treble range for this input.
put_Bass	Sets the bass equalization for this input.
get_Bass	Retrieves the bass equalization for this input.
get_BassRange	Retrieves the bass range for this input.

IAMAudioInputMixer::get_Bass

IAMAudioInputMixer Interface

Retrieves the bass equalization for this input.

HRESULT get_Bass(
  double **pBass*
  );

Parameters

pBass
    [in] Pointer to the bass gain in decibels (a negative value means attenuate).

Return Values

Returns an HRESULT value that depends on the implementation of the interface. **HRESULT** can include one of the following standard constants, or other values not listed.

Value	Meaning
E_FAIL	Failure.
E_POINTER	Null pointer argument.
E_INVALIDARG	Invalid argument.
E_NOTIMPL	Method isn't supported.
NOERROR	Success.

IAMAudioInputMixer::get_BassRange

IAMAudioInputMixer Interface

Retrieves the bass range for this input.

HRESULT get_BassRange(
  double **pRange*
  );

Parameters

pRange
  [out, retval] Largest value allowed in the bass range specified in put_Bass. For example, 6.0 means any value between -6.0 and 6.0 is allowed.

Return Values

Returns an HRESULT value that depends on the implementation of the interface. **HRESULT** can include one of the following standard constants, or other values not listed.

Value	**Meaning**
E_FAIL	Failure.
E_POINTER	Null pointer argument.
E_INVALIDARG	Invalid argument.
E_NOTIMPL	Method isn't supported.
NOERROR	Success.

IAMAudioInputMixer::get_Enable

IAMAudioInputMixer Interface

Retrieves whether the input is enabled.

HRESULT get_Enable(
  BOOL **pfEnable*

);

Parameters

pfEnable
: [in] Pointer to a value indicating whether mixing is enabled for the input. TRUE indicates the input is enabled, FALSE indicates the input is disabled.

Return Values

Returns an HRESULT value that depends on the implementation of the interface. **HRESULT** can include one of the following standard constants, or other values not listed.

Value	Meaning
E_FAIL	Couldn't retrieve information.
E_POINTER	Null pointer argument.
E_INVALIDARG	Invalid argument.
E_NOTIMPL	Method isn't supported.
NOERROR	Success.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

IAMAudioInputMixer::get_Loudness

IAMAudioInputMixer Interface

Retrieves the loudness control setting for this input.

HRESULT get_Loudness(
  int **pfLoudness*
  );

Parameters

pfLoudness
: [in] Pointer to value indicating whether loudness is on or off. TRUE indicates on, FALSE indicates off.

Return Values

Returns an HRESULT value that depends on the implementation of the interface. **HRESULT** can include one of the following standard constants, or other values not listed.

Value	Meaning
E_FAIL	Error.
E_POINTER	Null pointer argument.
E_INVALIDARG	Invalid argument.
E_NOTIMPL	Method isn't supported.
E_OUTOFMEMORY	Out of memory.
NOERROR	Success.

Previous Home Topic Contents Index Next

IAMAudioInputMixer::get_MixLevel

IAMAudioInputMixer Interface

Retrieves the recording level for this input.

HRESULT get_MixLevel(
  double **pLevel*
  );

Parameters

[out] pLevel
Pointer to the value of the recording level. Values range between 0 (off) and 1 (full volume). AMF_AUTOMATICGAIN (-0x0001), if supported, means automatic adjustment of level.

Return Values

Returns an HRESULT value that depends on the implementation of the interface. **HRESULT** can include one of the following standard constants, or other values not listed.

Value	Meaning
E_FAIL	Error retrieving recording level.
E_POINTER	Null pointer argument.
E_INVALIDARG	Invalid argument.
E_NOTIMPL	Method isn't supported.
NOERROR	Success.

IAMAudioInputMixer::get_Mono

IAMAudioInputMixer Interface

Retrieves whether all channels of an input are combined into a mono signal.

HRESULT get_Mono(
  BOOL **pfMono*
  );

Parameters

pfMono
  [in] Pointer to a value indicating whether mono is enabled. TRUE indicates mono is enabled, FALSE indicates mono is disabled.

Return Values

Returns an HRESULT value that depends on the implementation of the interface. **HRESULT** can include one of the following standard constants, or other values not listed.

Value	**Meaning**
E_FAIL	Error getting mono control.
E_POINTER	Null pointer argument.
E_INVALIDARG	Invalid argument.
E_NOTIMPL	Method isn't supported.
NOERROR	Success.

IAMAudioInputMixer::get_Pan

IAMAudioInputMixer Interface

Retrieves the pan level for this input.

HRESULT get_Pan(
  double * *pPan*
  );

Parameters

pPan
  [in] Pointer to the value of the pan level. Possible levels are from -1 to 1, with specific values as follows:

Value	**Meaning**
-1	Full left
0	Center
1	Full right

Return Values

Returns an HRESULT value that depends on the implementation of the interface. **HRESULT** can include one of the following standard constants, or other values not listed.

Value	**Meaning**
E_FAIL	Error retrieving pan level.
E_POINTER	Null pointer argument.
E_INVALIDARG	Invalid argument.
E_NOTIMPL	Can't pan: not stereo.
NOERROR	Success.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

IAMAudioInputMixer::get_Treble

IAMAudioInputMixer Interface

Retrieves the treble equalization for this input.

HRESULT get_Treble(
  double **pTreble*
  );

Parameters

pTreble

[in] Pointer to the treble gain in decibels (a negative value means attenuate).

Return Values

Returns an HRESULT value that depends on the implementation of the interface. **HRESULT** can include one of the following standard constants, or other values not listed.

Value	**Meaning**
E_FAIL	Failure.
E_POINTER	Null pointer argument.
E_INVALIDARG	Invalid argument.
E_NOTIMPL	Method isn't supported.
NOERROR	Success.

IAMAudioInputMixer::get_TrebleRange

IAMAudioInputMixer Interface

Retrieves the treble range for this input.

HRESULT get_TrebleRange(
  double **pRange*
  );

Parameters

pRange
: [out, retval] Largest value allowed in the treble range. This is the maximum value allowed in put_Treble. For example, 6.0 means any value between -6.0 and 6.0 is allowed.

Return Values

Returns an HRESULT value that depends on the implementation of the interface. **HRESULT** can include one of the following standard constants, or other values not listed.

Value	Meaning
E_FAIL	Failure.
E_POINTER	Null pointer argument.
E_INVALIDARG	Invalid argument.
E_NOTIMPL	Method isn't supported.
NOERROR	Success.

Previous Home Topic Contents Index Next

IAMAudioInputMixer::put_Bass

IAMAudioInputMixer Interface

Sets the bass equalization for this input.

HRESULT put_Bass(
 double *Bass*
 );

Parameters

Bass
 [in] Gain in decibels (a negative value means attenuate).

Return Values

Returns an HRESULT value that depends on the implementation of the interface. **HRESULT** can include one of the following standard constants, or other values not listed.

Value	Meaning
E_FAIL	Failure.
E_POINTER	Null pointer argument.
E_INVALIDARG	Argument is invalid. Must be in range given by get_BassRange.
E_NOTIMPL	Method isn't supported.
NOERROR	Success.

Remarks

Boosts or cuts the signal's bass before it is recorded by the number of decibels specified by *Bass*.

Previous Home Topic Contents Index Next

IAMAudioInputMixer::put_Enable

IAMAudioInputMixer Interface

Enables or disables an input in the mix.

HRESULT put_Enable(
BOOL *fEnable*
);

Parameters

fEnable
[in] Value to enable or disable an input. TRUE enables the input, FALSE disables it.

Return Values

Returns an HRESULT value that depends on the implementation of the interface. **HRESULT** can include one of the following standard constants, or other values not listed.

Value	**Meaning**
E_FAIL	Failed to enable or disable an input.
E_POINTER	Null pointer argument.
E_INVALIDARG	Invalid argument.
E_NOTIMPL	Method isn't supported.
NOERROR	Successfully enabled or disabled an input.

Remarks

If disabled, this input will not be mixed in as part of the recorded signal.

Previous Home Topic Contents Index Next

IAMAudioInputMixer::put_Loudness

IAMAudioInputMixer Interface

Turns the loudness control for this input on or off.

HRESULT put_Loudness(
  BOOL *fLoudness*
  );

Parameters

fLoudness
  [in] TRUE sets loudness on, FALSE sets loudness off.

Return Values

Returns an HRESULT value that depends on the implementation of the interface. **HRESULT** can include one of the following standard constants, or other values not listed.

Value	Meaning
E_FAIL	Loudness control set.
E_POINTER	Null pointer argument.
E_INVALIDARG	Invalid argument.
E_NOTIMPL	Method isn't supported.
NOERROR	Success.

Remarks

IAMAudioInputMixer::put_Loudness boosts the bass of low volume signals before they are recorded to compensate for the fact that your ear doesn't hear quiet bass sounds as well as other sounds.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

IAMAudioInputMixer::put_MixLevel

IAMAudioInputMixer Interface

Sets the record level for this input.

HRESULT put_MixLevel(
  double *Level*

```
  );
```

Parameters

Level
: Recording level. Values range between 0 (off) and 1 (full volume). AMF_AUTOMATICGAIN (-0x0001), if supported, means automatic adjustment of level.

Return Values

Returns an HRESULT value that depends on the implementation of the interface. **HRESULT** can include one of the following standard constants, or other values not listed.

Value	Meaning
E_FAIL	Error setting volume.
E_POINTER	Null pointer argument.
E_INVALIDARG	Record level must be between 0 and 1.
E_NOTIMPL	Automatic gain currently not implemented.
NOERROR	Success.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

IAMAudioInputMixer::put_Mono

IAMAudioInputMixer Interface

Combines all channels of an input into a mono signal.

HRESULT put_Mono(
  BOOL *fMono*
  );

Parameters

fMono
: [in] TRUE indicates mono, FALSE indicates multichannel.

Return Values

Returns an HRESULT value that depends on the implementation of the interface. **HRESULT** can include one of the following standard constants, or other values not listed.

Value	Meaning
E_FAIL	Error setting mono control.
E_POINTER	Null pointer argument.
E_INVALIDARG	Invalid argument.
E_NOTIMPL	Method isn't supported.
NOERROR	Success.

Remarks

When set to mono mode, making a stereo recording of this input will have both channels contain the same data. The result will be a mixture of the left and right signals.

IAMAudioInputMixer::put_Pan

IAMAudioInputMixer Interface

Sets the pan for this input.

HRESULT put_Pan(
 double *Pan*
 );

Parameters

Pan
[in] Pan level. Possible values for *Pan* are from -1 to 1, with specific values as follows:

Value	Meaning
-1	Full left
0	Center
1	Full right

Return Values

Returns an HRESULT value that depends on the implementation of the interface. **HRESULT** can include one of the following standard constants, or other values not listed.

Value	**Meaning**
E_FAIL	Error setting volume.
E_POINTER	Null pointer argument.
E_INVALIDARG	Pan level must be between -1 and 1.
E_NOTIMPL	Can't pan: not stereo.
NOERROR	Success.

Remarks

Setting the pan of an input to full left makes that input's signal go only into the left channel of a stereo recording. Panning has no effect for a mono recording.

IAMAudioInputMixer::put_Treble

IAMAudioInputMixer Interface

Sets the treble equalization for this input.

HRESULT put_Treble(
  [in] double *Treble*
  );

Parameters

Treble
    [in] Gain in decibels (a negative value means attenuate).

Return Values

Returns an HRESULT value that depends on the implementation of the interface. **HRESULT** can include one of the following standard constants, or other values not listed.

Value	**Meaning**
E_FAIL	Failure.
E_POINTER	Null pointer argument.
E_INVALIDARG	Argument is invalid. Must be in range given by get_TrebleRange.
E_NOTIMPL	Method isn't supported.
NOERROR	Success.

Remarks

This method boosts or cuts the signal's treble by a specified number of decibels before it is recorded.

IAMBufferNegotiation Interface

The **IAMBufferNegotiation** interface tells a pin what kind of buffer specifications it should use when connected. Use this interface when an application requires control over allocating the number of buffers that pins will use when transporting media samples between filters.

The IAMBufferNegotiation::SuggestAllocatorProperties method accepts an ALLOCATOR_PROPERTIES structure that contains the allocator's count, size, alignment, and prefix properties that you want to use. Typically, you set only the *cBuffers* member of the **ALLOCATOR_PROPERTIES** structure, which refers to the number of buffers at the specified allocator. All other properties should indicate a negative number to enable your capture hardware to use its own default values.

If a negative value is specified for *cBuffers*, the allocator will try to allocate as many buffers as it needs, which depends on the available resources and capture frame rate. If you specify a higher value, the allocator will try to allocate more buffers, up to the system's available memory. Allocating a lower number of buffers can result in dropped frames. For teleconferencing applications, it may be desirable to set this number to a smaller value (for example, 2 is a reasonable setting if the network can only support transmission of 2 frames per second (fps) at a given video format and resolution).

Applications can call the IAMBufferNegotiation::GetAllocatorProperties method to retrieve the properties of the allocator being used.

When to Implement

Implement this interface when your pin will connect to another pin by using the IMemInputPin interface and you want to enable an application to allocate the buffer settings to be used for transporting media samples between filters. All capture filters should support this interface to enable applications to specify precise settings for buffers (see Vidcap.cpp and Vidcap.h in the \Samples\DS\Vidcap directory for a sample implementation).

When to Use

Teleconferencing applications should use this interface to specify a minimal number of buffers. This tells the capture filter not to waste resources buffering information in slower capture or disk-writing scenarios.

Methods in Vtable Order

IUnknown methods	**Description**
QueryInterface	Retrieves pointers to supported interfaces.
AddRef	Increments the reference count.
Release	Decrements the reference count.

IAMBufferNegotiation methods	**Description**
SuggestAllocatorProperties	Asks a pin to use the allocator buffer properties set in the ALLOCATOR_PROPERTIES structure.
GetAllocatorProperties	Retrieves the properties of the allocator being used by a pin.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

IAMBufferNegotiation::GetAllocatorProperties

IAMBufferNegotiation Interface

Retrieves the properties of the allocator that a pin is using.

HRESULT GetAllocatorProperties(
  ALLOCATOR_PROPERTIES **pprop*
  );

Parameters

pprop
[out] Pointer to an ALLOCATOR_PROPERTIES structure.

Return Values

Returns an HRESULT value that depends on the implementation of the interface.

Remarks

Call this method only after the pins connect.

See Also

SuggestAllocatorProperties

IAMBufferNegotiation::SuggestAllocatorProperti

IAMBufferNegotiation Interface

Asks a pin to use the allocator buffer properties set in the ALLOCATOR_PROPERTIES structure.

HRESULT SuggestAllocatorProperties(
 const ALLOCATOR_PROPERTIES **pprop*
 );

Parameters

pprop
 [in] Pointer to an ALLOCATOR_PROPERTIES structure.

Return Values

Returns an HRESULT value that depends on the implementation of the interface.

Remarks

An application must call this function before two pins are connected. If the pins are connected before you call this method, then the filter graph will have already negotiated the buffer and it will be too late for an application to preallocate them.

Applications must call this method on both pins being connected to ensure that the other pin doesn't overrule the application's request. However, if one pin doesn't support this interface, a single call will be sufficient.

Use a negative number for any element in the ALLOCATOR_PROPERTIES structure to set properties to default values.

See Also

IAMBufferNegotiation::GetAllocatorProperties

IAMCollection Interface

The filter graph manager exposes **IAMCollection**, which allows access to collections of objects such as those exporting IPinInfo and IFilterInfo interfaces.

When to Implement

This interface is implemented by the filter graph manager for use by Automation client applications, such as Microsoft® Visual Basic®.

When to Use

Applications that use Automation use this interface indirectly when retrieving collections of objects. For example, the IFilterInfo::get_Pins method retrieves an IAMCollection interface that can be used to access the IPinInfo interfaces corresponding to the pins on the filter.

Methods in Vtable Order

IUnknown methods	**Description**
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments the reference count.
Release	Decrements the reference count.
IDispatch methods	**Description**
GetTypeInfoCount	Determines whether there is type information available for this dispinterface.
GetTypeInfo	Retrieves the type information for this dispinterface if GetTypeInfoCount returned successfully.
GetIDsOfNames	Converts text names of properties and methods (including arguments) to their corresponding DISPIDs.
Invoke	Calls a method or accesses a property in this dispinterface if given a DISPID and any other necessary parameters.
IAMCollection methods	**Description**
get_Count	Retrieves the number of items in the collection.
get_NewEnum	Retrieves an enumerator object that implements IEnumVARIANT on this collection.
Item	Retrieves the indexed item from the collection.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

IAMCollection::get_Count

IAMCollection Interface

Retrieves the number of items in the collection.

HRESULT get_Count(
 LONG **plCount*
 );

Parameters

plCount
 [out, retval] Number of items in the collection.

Return Values

Returns an HRESULT value.

IAMCollection::get_NewEnum

IAMCollection Interface

Retrieves an enumerator object that implements IEnumVARIANT on this collection.

HRESULT get_NewEnum(
 IUnknown ***ppUnk*
 );

Parameters

ppUnk
 [out, retval] IUnknown for an object that implements IEnumVARIANT on this collection.

Return Values

Returns an HRESULT value.

IAMCollection::Item

IAMCollection Interface

Retrieves the indexed item from the collection.

HRESULT Item(
  long *lItem*,
  IUnknown ***ppUnk*
  );

Parameters

lItem
  [in] Index into the collection.
ppUnk
  [out] Returned IUnknown interface for the contained item.

Return Values

Returns an HRESULT value.

Remarks

The returned *ppUnk* parameter represents an object corresponding to the type of objects in the container. It can be an IFilterInfo, IPinInfo, or IMediaTypeInfo object. The index is zero-based.

Previous Home Topic Contents Index Next

IAMCopyCaptureFileProgress Interface

The **IAMCopyCaptureFileProgress** interface contains one method, Progress, which the ICaptureGraphBuilder::CopyCaptureFile method can call to receive information on the percentage complete of a copy operation.

When to Implement

Capture applications implement this method when they need to receive information on the percentage complete of a copy operation.

When to Use

Use this interface when applications need to check the copying progress of a captured file.

Methods in Vtable Order

IUnknown methods	Description
QueryInterface	Retrieves pointers to supported interfaces.
AddRef	Increments the reference count.
Release	Decrements the reference count.

IAMCopyCaptureFileProgress methods	Description
Progress	Sends applications the progress (percentage complete) of a copy operation that the ICaptureGraphBuilder::CopyCaptureFile method is performing.

IAMCopyCaptureFileProgress::Progress

IAMCopyCaptureFileProgress Interface

Sends applications the progress (percentage complete) of a copy operation that the ICaptureGraphBuilder::CopyCaptureFile method is performing.

HRESULT Progress(
  int *iProgress*
  );

Parameters

iProgress
[in] Percentage of copy complete between 0 and 100.

Return Values

Returns S_OK if successful or S_FALSE if the operation is aborted.

Remarks

The ICaptureGraphBuilder::CopyCaptureFile can call this method to inform applications of the copy operation's progress.

This method is called periodically while ICaptureGraphBuilder::CopyCaptureFile is running.

IAMCrossbar Interface

The **IAMCrossbar** interface is exposed on analog video crossbar filters and is used to route messages from an analog or digital audio or video source to a video capture filter. The crossbar filter is modeled after a general switching matrix, with *n* inputs and *m* outputs. Any of the input signals can be routed to one or more of the outputs.

A single crossbar can route both video and audio signals. You can also use a video pin to route only the audio portion of a combined signal.

This filter is based on a simple multiplexer.

When to Implement

Implement this interface when your filter needs to route analog or digital signals to a capture filter.

When to Use

Use this interface when your application needs to route analog or digital video signals through a crossbar filter.

Methods in Vtable Order

IUnknown methods	Description
QueryInterface	Retrieves pointers to supported interfaces.
AddRef	Increments the reference count.
Release	Decrements the reference count.

IAMCrossbar methods	Description
get_PinCounts	Retrieves the number of input and output pins.
CanRoute	Determines if the crossbar filter can route the analog or digital signal.
Route	Routes an input pin to an output pin.
get_IsRoutedTo	Retrieves the input pin connected to a given output pin.
get_CrossbarPinInfo	Retrieves a pin that has audio or video data relating to a given pin.

Previous Home Topic Contents Index Next

IAMCrossbar::CanRoute

IAMCrossbar Interface

Determines if routing is possible.

HRESULT CanRoute (
  long *OutputPinIndex*,
  long *InputPinIndex*
  );

Parameters

OutputPinIndex
	[in] Output pin.
InputPinIndex
	[in] Input pin.

Return Values

Returns an HRESULT value that depends on the implementation of the interface.

IAMCrossbar::get_CrossbarPinInfo

IAMCrossbar Interface

Retrieves a pin that has audio or video data relating to a given pin.

HRESULT get_CrossbarPinInfo (
  BOOL *IsInputPin*,
  long *PinIndex*,
  long * *PinIndexRelated*,
  long * *PhysicalType*
  );

Parameters

IsInputPin
	[in] Specify TRUE for an input pin; FALSE for an output pin.

PinIndex
    [in] Pin to find a related pin for.
PinIndexRelated
    [out] Index value of the related pin.
PhysicalType
    [out] Physical type of pin (audio or video).

Return Values

Returns an HRESULT value that depends on the implementation of the interface.

Remarks

This method retrieves, for example, the audio pin related to a given video pin.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

IAMCrossbar::get_IsRoutedTo

IAMCrossbar Interface

Retrieves the input pin connected to a given output pin.

HRESULT get_IsRoutedTo (
  long *OutputPinIndex*,
  long * *InputPinIndex*
  );

Parameters

OutputPinIndex
    [in] Output pin.
InputPinIndex
    [out] Pointer to the connected input pin.

Return Values

Returns an HRESULT value that depends on the implementation of the interface.

Previous Home Topic Contents Index Next

IAMCrossbar::get_PinCounts

IAMCrossbar Interface

Retrieves the number of input and output pins.

HRESULT get_PinCounts(
  long * *OutputPinCount*,
  long * *InputPinCount*
  );

Parameters

OutputPinCount
 [out] Number of output pins.
InputPinCount
 [out] Number of input pins.

Return Values

Returns an HRESULT value that depends on the implementation of the interface.

IAMCrossbar::Route

IAMCrossbar Interface

Routes an input pin to an output pin.

HRESULT Route (
  long *OutputPinIndex*,
  long *InputPinIndex*
  );

Parameters

OutputPinIndex
 [in] Output pin.
InputPinIndex
 [in] Input pin.

Return Values

Returns an HRESULT value that depends on the implementation of the interface.

Remarks

Pin indexes are zero based.

IAMCutListElement Interface

The **IAMCutListElement** interface describes a base object, which represents an element in a cutlist. For a simpler interface that provides basic cutlist functionality, applications can use IFileClip to create an object that supports this interface.

See About Cutlists and Using Cutlists for more information.

When to Implement

Usually, you don't need to implement this interface because DirectShow provides the CLSID_VideoFileClip and CLSID_AudioFileClip objects, which can create an object that implements it for you. However, you can implement this interface in your application when you need to change this interface's default behavior.

When to Use

Use this interface in your filter when you need to get specific elements of a cutlist.

When compiling a cutlist application you must explicitly include the cutlist header file as follows:

```
#include <cutlist.h>
```

Methods in Vtable Order

IUnknown methods	**Description**
QueryInterface	Retrieves pointers to supported interfaces.
AddRef	Increments the reference count.
Release	Decrements the reference count.

IAMCutListElement methods	Description
GetElementStartPosition	Retrieves the media time of the element's start in the time scale of the cutlist.
GetElementDuration	Retrieves the duration of the cutlist element.
IsFirstElement	Determines if the element is the first in the cutlist.
IsLastElement	Determines if the element is the last in the cutlist.
IsNull	Determines if the element is null.
ElementStatus	Determines the status of the element.

IAMCutListElement::ElementStatus

IAMCutListElement Interface

Determines the status of the element.

HRESULT ElementStatus(
  DWORD **pdwStatus*,
  DWORD *dwTimeoutMs*
  );

Parameters

pdwStatus
[in/out] Status. On input, if this parameter contains CL_WAIT_FOR_STATE and an additional state value from the CL_ELEM_STATUS enumerated data type, this method waits *dwTimeoutMs* milliseconds until the element is in that state before returning. On output, this is a logical combination of flags from the **CL_ELEM_STATUS** enumerated data type.

dwTimeoutMs
[in] Timeout value, in milliseconds.

Return Values

Returns an HRESULT value that depends on the implementation of the interface. **HRESULT** can include one of the following standard constants, or other values not listed.

Value	Meaning
E_FAIL	Failure.
E_INVALIDARG	Argument is invalid.
E_NOTIMPL	Method is not supported.
S_OK	Success. The element is null.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

IAMCutListElement::GetElementDuration

IAMCutListElement Interface

Retrieves the duration of the cutlist element.

HRESULT GetElementDuration(
  REFERENCE_TIME **pmtDuration*
  );

Parameters

pmtDuration
  [out] Duration of the element in REFERENCE_TIME.

Return Values

Returns an HRESULT value that depends on the implementation of the interface. **HRESULT** can include one of the following standard constants, or other values not listed.

Value	Meaning
E_FAIL	Failure.
E_INVALIDARG	Argument is invalid.
E_NOTIMPL	Method is not supported.
E_POINTER	Null pointer argument.
S_OK	Success.

Remarks

When you call the IFileClip::CreateCut method to create the element, the difference between its *mtTrimOut* and *mtTrimIn* parameters determines the duration.

Previous Home Topic Contents Index Next

IAMCutListElement::GetElementStartPosition

IAMCutListElement Interface

Retrieves the media time of the element's start in the time scale of the cutlist.

HRESULT GetElementStartPosition(
  REFERENCE_TIME **pmtStart*
  );

Parameters

pmtStart
 [out] Pointer to the media time for the start of the element.

Return Values

Returns an HRESULT value that depends on the implementation of the interface. **HRESULT** can include one of the following standard constants, or other values not listed.

Value	Meaning
E_FAIL	Failure.
E_INVALIDARG	Argument is invalid.
E_NOTIMPL	Method is not supported.
E_POINTER	Null pointer argument.
S_OK	Success.

Remarks

Times retrieved by this method are relative to the time within the cutlist. For example, the first element in the cutlist starts at time zero.

Previous Home Topic Contents Index Next

IAMCutListElement::IsFirstElement

IAMCutListElement Interface

Determines if the element is the first in the cutlist.

HRESULT IsFirstElement(void);

Return Values

Returns an HRESULT value that depends on the implementation of the interface. **HRESULT** can include one of the following standard constants, or other values not listed.

Value	Meaning
E_FAIL	Failure.
E_INVALIDARG	Argument is invalid.
E_NOTIMPL	Method is not supported.
S_OK	Success. This is the first element in the cutlist.

IAMCutListElement::IsLastElement

IAMCutListElement Interface

Determines if the element is the last in the cutlist.

HRESULT IsLastElement(void);

Return Values

Returns an HRESULT value that depends on the implementation of the interface. **HRESULT** can include one of the following standard constants, or other values not listed.

Value	Meaning
E_FAIL	Failure.
E_INVALIDARG	Argument is invalid.
E_NOTIMPL	Method is not supported.
S_OK	Success. This is the last element in the cutlist.

IAMCutListElement::IsNull

IAMCutListElement Interface

Determines if the element is null.

HRESULT IsNull(void);

Return Values

Returns an HRESULT value that depends on the implementation of the interface. **HRESULT** can include one of the following standard constants, or other values not listed.

Value	Meaning
E_FAIL	Failure.
E_INVALIDARG	Argument is invalid.
E_NOTIMPL	Method is not supported.
S_FALSE	Element is not null.
S_OK	Success. The element is null.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

IAMDevMemoryAllocator Interface

The **IAMDevMemoryAllocator** interface enables the creation of third-party memory allocators by using an on-board memory manager object. Many codec hardware manufacturers put on-board mapped memory for the codecs to improve the efficiency of buffer manipulation. This interface allocates that memory and provides the GetDevMemoryObject method to retrieve a device memory control object, which supports the IAMDevMemoryControl interface. Devices that share the same device ID can use the memory.

The global memory manager object exposes this interface to allocate memory from memory that is on a particular device.

When to Implement

Implement this interface when your pin must support the creation of on-board memory allocators. Source filters that are aware of on-board memory and need to create their own allocators should query for this interface, request an amount of memory and then create an allocator (aggregating the device memory control object). Source filters that don't need to create their own allocator could just use the allocator of the downstream pin (which also aggregates the device memory control object). The hardware-based filter can confirm the usage of its on-board memory by calling methods on the aggregated allocator.

When to Use

Use this interface when applications need to control the memory of codecs with on-board memory.

Methods in Vtable Order

IUnknown methods	Description
QueryInterface	Retrieves pointers to supported interfaces.
AddRef	Increments the reference count.
Release	Decrements the reference count.

IAMDevMemoryAllocator methods	Description
GetInfo	Retrieves information about the memory capabilities.
CheckMemory	Tests whether a memory pointer was allocated by the specific instance (device) of the allocator.
Alloc	Allocates a memory buffer.
Free	Frees the previously allocated memory.
GetDevMemoryObject	Retrieves an IUnknown interface pointer to a device memory control object that can be aggregated with a custom allocator.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

IAMDevMemoryAllocator::Alloc

IAMDevMemoryAllocator Interface

Allocates a memory buffer.

HRESULT Alloc(
  BYTE ***ppBuffer**,
  DWORD **pdwcbBuffer*
  );

Parameters

ppBuffer

[out] Address of a pointer to the allocated memory buffer.

pdwcbBuffer

[in, out] For input, the number of bytes to allocate. For output, the number of actual bytes allocated.

Return Values

Returns S_OK if the desired quantity of memory was allocated, S_FALSE if memory was unavailable.

Remarks

Call this method to allocate a block of memory from the available pool.

See Also

IAMDevMemoryAllocator::Free

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

IAMDevMemoryAllocator::CheckMemory

IAMDevMemoryAllocator Interface

Tests whether a memory pointer was allocated by the specific instance (device) of the allocator.

HRESULT CheckMemory(
  const BYTE **pBuffer*
  );

Parameters

pBuffer

[in] Pointer to the allocated memory buffer's address.

Return Values

Returns S_OK if the on-board allocator allocated the memory, or S_FALSE if not. Memory that is on the particular device but not allocated will also return S_FALSE.

Remarks

The hardware filter typically uses this method to test whether the pointer actually points to on-board memory.

IAMDevMemoryAllocator::Free

IAMDevMemoryAllocator Interface

Frees the previously allocated memory.

HRESULT Free(
  BYTE **pBuffer*
  );

Parameters

pBuffer
 [in] Pointer to the allocated memory.

Return Values

Returns E_INVALIDARG if the specified allocator didn't allocate the memory (that is, CheckMemory fails).

Remarks

This method frees a block of memory from the pool.

IAMDevMemoryAllocator::GetDevMemoryObject

IAMDevMemoryAllocator Interface

Retrieves an IUnknown interface pointer to a device memory control object that can be aggregated with a custom allocator.

HRESULT GetDevMemoryObject(
  IUnknown ***ppUnkInnner*,
  IUnknown **pUnkOuter*
  );

Parameters

ppUnkInnner
[out] Address of a pointer to the newly created control object's own IUnknown. This inner **IUnknown** interface should be released when the outer object is destroyed. The custom allocator should call the QueryInterface method on this pointer to obtain the IAMDevMemoryControl interface.

pUnkOuter
[in] Pointer to the custom allocator's own IUnknown interface. This interface aggregates the device memory control object inside the custom allocator.

Return Values

Returns an HRESULT value that depends on the implementation of the interface.

Remarks

The device memory control object is necessary to aggregate with the custom allocator, because renderers that require the use of on-board memory will query for IAMDevMemoryControl when they receive a new allocator, to verify that the memory is from the same device. This occurs because the hardware filter will receive an IMemAllocator object, which might or might not use the on-board memory. To decide if it is a compatible allocator, the object would query for the **IAMDevMemoryControl** interface to access specific methods. The **IAMDevMemoryControl** creates an aggregated object that implements the methods of **IAMDevMemoryControl** (these are often hardware-specific).

See COM documentation for rules on how the outer object implements aggregation.

Previous Home Topic Contents Index Next

IAMDevMemoryAllocator::GetInfo

IAMDevMemoryAllocator Interface

Retrieves information about the memory capabilities.

HRESULT GetInfo(
  DWORD **pdwcbTotalFree*,
  DWORD **pdwcbLargestFree*,
  DWORD **pdwcbTotalMemory*,
  DWORD **pdwcbMinimumChunk*

```
);
```

Parameters

pdwcbTotalFree
 [out] Total free memory size.
pdwcbLargestFree
 [out] Retrieves the largest free memory size.
pdwcbTotalMemory
 [out] Retrieves the total memory size.
pdwcbMinimumChunk
 [out] Retrieves the minimum chunk size, giving granularity and alignment rules.

Return Values

Returns an HRESULT value that depends on the implementation of the interface.

Remarks

Use this method to find out the total amount of memory available. This method returns values for the entire on-board memory that is available on that device. If multiple filters (devices) share the memory, it will return the amount available to that specific device, which might be a portion of the total amount of on-board memory. This amount will be implementation-specific. For example, the on-board memory manager on the codec might be able to access all 32 megabytes (MB) of memory on the card. However, individual pin implementations of IAMDevMemoryAllocator only report a portion of this memory.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

IAMDevMemoryControl Interface

The **IAMDevMemoryControl** interface controls and identify the on-board memory of codecs. A device memory control object supports this interface. This object is aggregated with an IMemAllocator object that is used in the connection. Typically, filters will call the IAMDevMemoryAllocator::GetDevMemoryObject method to obtain a pointer to this interface.

When to Implement

Implement this interface with the IAMDevMemoryAllocator interface when pins need to have greater control of memory allocation.

When to Use

Use this interface to synchronize the completed data write of a memory allocator, and to get the device ID of the on-board memory allocator.

Methods in Vtable Order

IUnknown methods	**Description**
QueryInterface	Retrieves pointers to supported interfaces.
AddRef	Increments the reference count.
Release	Decrements the reference count.

IAMDevMemoryControl methods	**Description**
QueryWriteSync	Checks if the memory supported by the allocator requires the use of the WriteSync method.
WriteSync	Used to synchronize with the completed write. This method returns when any data being written to the specified allocator region is fully written into the memory.
GetDevId	Retrieves the device ID of the on-board memory allocator.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

IAMDevMemoryControl::GetDevId

IAMDevMemoryControl Interface

Retrieves the device ID of the on-board memory allocator.

HRESULT GetDevId(
  DWORD **pdwDevId*
  );

Parameters

pdwDevId
    [out] Pointer to the device ID.

Return Values

Returns an HRESULT value that depends on the implementation of the interface.

Remarks

This method retrieves a unique ID that the hardware filter can use to verify that the specified allocator passed uses its on-board memory (because there can be more than one). The ID will be the same one as used to create the allocator object (using **CoCreateNamedInstance**). For another filter to be able to use the on-board memory, it must have the same device ID as the on-board memory allocator.

Previous Home Topic Contents Index Next

IAMDevMemoryControl::QueryWriteSync

IAMDevMemoryControl Interface

Checks if the memory supported by the allocator requires the use of the IAMDevMemoryControl::WriteSync method.

HRESULT QueryWriteSync( );

Return Values

Returns S_OK if the method is required, or S_FALSE otherwise.

Remarks

Not all on-board memory needs to have WriteSync called to synchronize with the completed write. This method is used to check if the call is necessary.

Previous Home Topic Contents Index Next

IAMDevMemoryControl::WriteSync

IAMDevMemoryControl Interface

Used to synchronize with the completed write. This method returns when any data being written to the particular allocator region is fully written into the memory.

HRESULT WriteSync( );

Return Values

Returns an HRESULT value that depends on the implementation of the interface. Common return values include:

Value	Meaning
E_FAIL	A time-out has occurred without confirming that data was written.
S_OK	Operation proceeded normally.
VFW_E_NOT_COMMITTED	The allocator hasn't called the IMemAllocator::Commit method.

Remarks

This method guarantees that all prior write operations to allocated memory have succeeded. Subsequent memory write operations require another call to **WriteSync**.

This method is implementation dependent, and is used (when necessary) to synchronize memory write operations to the memory. The driver of the on-board memory provides the implementation.

The IAMDevMemoryControl interface is typically found on memory that is accessed through a PCI-bridge. Memory behind a PCI bridge must be synchronized after a memory write operation completes if another device will access that memory from behind the PCI bridge. This is because the host access to the memory is buffered via the PCI bridge FIFO (first in first out), and the host will think the write is completed before the bridge actually writes the data. A subsequent action by a device behind the bridge, such as a SCSI controller, might read the memory before the write is completed if the **IAMDevMemoryControl::WriteSync** method is not called.

Previous Home Topic Contents Index Next

IAMDirectSound Interface

The **IAMDirectSound** interface provides access from Microsoft® DirectShow™ to Microsoft DirectX™ audio interfaces, such as IDirectSound and IDirectSoundBuffer. This enables you to play back the audio portions of DirectShow-compatible media files anywhere within the 3-D space of a DirectX application, making your applications much more absorbing and lifelike.

After you connect the media source file to a sound renderer on a filter graph, you can use DirectSound's functionality to position or manipulate the sound playback as needed. For more information on the relevant DirectSound interfaces and methods, see the DirectX SDK documentation. After you finish with an interface you obtained through **IAMDirectSound**, be sure to release it by calling the appropriate method. If you disconnect the sound renderer from the graph before releasing the interfaces, your application might fail.

The DSound Audio Renderer filter implements this interface.

Note Only the GetWindowFocus and SetWindowFocus methods are currently implemented for this interface.

When to Implement

This interface is implemented by the DSound Audio Renderer filter.

When to Use

The DSound Audio Renderer filter uses this interface; it is not intended for other uses.

Methods in Vtable Order

IUnknown methods	**Description**
QueryInterface	Returns pointers to supported interfaces.
AddRef	Increments the reference count.
Release	Decrements the reference count.

IAMDirectSound methods	**Description**
GetDirectSoundInterface	Retrieves a handle to the current sound device's IDirectSound interface. Not currently implemented.
GetPrimaryBufferInterface	Retrieves a handle to the current sound device's primary sound buffer. Not currently implemented.
GetSecondaryBufferInterface	Retrieves a handle to the current sound device's secondary sound buffer. Not currently implemented.
ReleaseDirectSoundInterface	Releases the current sound device's IDirectSound interface. Not currently implemented.
ReleasePrimaryBufferInterface	Releases the current sound device's primary sound buffer. Not currently implemented.
ReleaseSecondaryBufferInterface	Releases the current sound device's secondary sound buffer. Not currently implemented.
SetWindowFocus	Sets the window that will handle sound playback for the current media file.
GetWindowFocus	Retrieves the window that is handling sound playback for the current media file.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

IAMDirectSound::GetDirectSoundInterface

IAMDirectSound Interface

Retrieves a handle to the current sound device's IDirectSound interface. Not currently implemented.

HRESULT IAMDirectSound::GetDirectSoundInterface(
  LPDIRECTSOUND **lplpds*

);

Parameters

lplpds
: Address of a pointer to an IDirectSound interface that will point to the current sound device's interface.

Return Values

Returns one of the following values.

Value	Meaning
E_FAIL	No sound device is available.
E_INVALIDARG	The *lplpds* parameter is null.
E_NOTIMPL	DirectSound isn't installed.
NOERROR	The method succeeded.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

IAMDirectSound::GetPrimaryBufferInterface

IAMDirectSound Interface

Retrieves a handle to the current sound device's primary sound buffer. Not currently implemented.

HRESULT IAMDirectSound::GetDirectSoundInterface(
  LPDIRECTSOUNDBUFFER **lplpdsb*
  );

Parameters

lplpdsb
: Address of a pointer to an IDirectSoundBuffer interface that will point to the current sound device's primary sound buffer.

Return Values

Returns one of the following values.

Value	Meaning
E_FAIL	No sound device is available.
E_INVALIDARG	The *lplpdsb* parameter is null.
E_NOTIMPL	DirectSound isn't installed.
NOERROR	The method succeeded.

IAMDirectSound::GetSecondaryBufferInterface

IAMDirectSound Interface

Retrieves a handle to the current sound device's secondary sound buffer. Not currently implemented.

HRESULT IAMDirectSound::GetSecondaryBufferInterface(
  LPDIRECTSOUNDBUFFER **lplpdsb*
  );

Parameters

lplpdsb
Address of a pointer to an IDirectSoundBuffer interface. On exit, it will point to the current sound device's secondary sound buffer.

Return Values

Returns one of the following values.

Value	Meaning
E_FAIL	No sound device is available.
E_INVALIDARG	The *lplpdsb* parameter is null.
E_NOTIMPL	DirectSound isn't installed.
NOERROR	The method succeeded.

IAMDirectSound::GetWindowFocus

IAMDirectSound Interface

Retrieves the window that is handling sound playback for the current media file.

HRESULT IAMDirectSound::GetWindowFocus(
  HWND* *hWnd*,
  BOOL *bMixingOnOrOff*
  );

Parameters

hWnd
: Handle to the sound playback window. If this value is null, the sound isn't associated with a window; note that Windows NT 4.0 does not currently support windowless sound playback.

bMixingOnOrOff
: Value indicating whether to mix the sound (TRUE) or not (FALSE).

Return Values

Returns one of the following values.

Value	**Meaning**
E_FAIL	No sound device is available.
E_INVALIDARG	The *hWnd* argument is invalid.
E_NOTIMPL	DirectSound isn't installed.
NOERROR	The method succeeded.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

IAMDirectSound::ReleaseDirectSoundInterface

IAMDirectSound Interface

Releases the current sound device's IDirectSound interface. Not currently implemented.

HRESULT IAMDirectSound::ReleaseDirectSoundInterface(
  LPDIRECTSOUND *lpds*

);

Parameters

lpds
Pointer to the IDirectSound interface to release.

Return Values

Returns one of the following values.

Value	Meaning
E_FAIL	There are no references to the specified IDirectSound interface, so it can't be released.
E_INVALIDARG	The *lpds* parameter is null.
NOERROR	The method succeeded.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

IAMDirectSound::ReleasePrimaryBufferInterface

IAMDirectSound Interface

Releases the current sound device's primary sound buffer. Not currently implemented.

HRESULT IAMDirectSound::ReleasePrimaryBufferInterface(
LPDIRECTSOUNDBUFFER *lpdsb*
);

Parameters

lpdsb
Pointer to the IDirectSoundBuffer interface to release.

Return Values

Returns one of the following values.

Value	Meaning
E_FAIL	There are no references to the specified IDirectSoundBuffer interface, so it can't be released.
E_INVALIDARG	The *lpdsb* parameter is null.
NOERROR	The method succeeded.

IAMDirectSound::ReleaseSecondaryBufferInterfa

IAMDirectSound Interface

Releases the current sound device's secondary sound buffer. Not currently implemented.

HRESULT IAMDirectSound::ReleaseSecondaryBufferInterface(
  LPDIRECTSOUNDBUFFER *lpdsb*
  );

Parameters

lpdsb
    Pointer to the IDirectSoundBuffer interface to release.

Return Values

Returns one of the following values.

Value	**Meaning**
E_FAIL	There are no references to the specified IDirectSoundBuffer interface, so it can't be released.
E_INVALIDARG	The *lpdsb* parameter is null.
NOERROR	The method succeeded.

IAMDirectSound::SetWindowFocus

IAMDirectSound Interface

Sets the window that will handle sound playback for the current media file.

HRESULT IAMDirectSound::SetWindowFocus(
  HWND *hWnd***,**
  BOOL *bMixingOnOrOff*
  );

Parameters

hWnd
  Handle to the sound playback window. If this value is null, the sound will not be associated with any window; note that Windows NT 4.0 does not currently support windowless sound playback.

bMixingOnOrOff
  Value indicating whether to mix the sound (TRUE) or not (FALSE).

Return Values

Returns one of the following values.

Value	Meaning
E_FAIL	No sound device is available.
E_INVALIDARG	The *hWnd* argument is invalid.
E_NOTIMPL	DirectSound isn't installed.
NOERROR	The method succeeded.

Previous Home Topic Contents Index Next

IAMDroppedFrames Interface

The **IAMDroppedFrames** interface provides information to an application from a capture filter about frames that the filter dropped (that is, did not send), the frame rate achieved (the length of time the graph ran divided by the number of frames not dropped), and the data rate achieved (the length of time the graph ran divided by the average frame size). A high number of dropped frames can detract from the smoothness of the video clip.

When to Implement

A capture filter's video output pin should always implement this interface.

When a capture filter runs, it sends frame numbers beginning with the sequence 0, 1, 2, 3 (numbers will be missing if frames were dropped). The time stamp of each frame sent corresponds to the filter graph clock's time when the image was digitized. The end time is the start time plus the video frame's duration.

Set the media time of each sample by using CMediaSample::SetMediaTime and using frame numbers for the start and end times. For example, the start-time and end-time sequence might appear as follows: (0,1) (1,2) (2,3). A downstream filter can easily tell that a frame was dropped by checking for gaps in the frame number sequence rather than by looking for gaps in the regular time stamps. The following start-time and end-time sequence reveals that frame number 3 was dropped: (1,2) (2,3) (4,5) (5,6).

Every time a capture filter goes from State_Stopped to State_Paused, it should reset all counts to zero.

If your filter runs, pauses, and then runs again, you must continue to deliver frames as if it never paused. The first frame after the second run can't be time stamped earlier than the last frame sent before the pause. That is, your filter must always increment the media time of each sample sent. Never send the same frame number twice, and never go back in time.

When to Use

Applications should use this interface all the time when capturing to update the current capture status. After capturing is done, applications should use this interface to determine the final capture results.

If you are using a WDM video capture filter, you can only query an output pin for this interface if the capture filter is connected to another filter in the graph.

Methods in Vtable Order

IUnknown methods	Description
QueryInterface	Retrieves pointers to supported interfaces.
AddRef	Increments the reference count.
Release	Decrements the reference count.

IAMDroppedFrames methods	Description
GetNumDropped	Retrieves the total number of frames that the pin dropped since it last started streaming.
GetNumNotDropped	Retrieves the total number of frames that the pin delivered downstream (did not drop).
GetDroppedInfo	Retrieves an array of frame numbers that were dropped.
GetAverageFrameSize	Retrieves the average size of frames that were not dropped.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

IAMDroppedFrames::GetAverageFrameSize

IAMDroppedFrames Interface

Retrieves the average size of frames that the pin dropped.

HRESULT GetAverageFrameSize(
  long * *plAverageSize* **);**

Parameters

plAverageSize
[out, retval] Average size of frames sent out the pin since the pin started streaming, in bytes.

Return Values

Returns an HRESULT value that depends on the implementation of the interface.

Previous Home Topic Contents Index Next

IAMDroppedFrames::GetDroppedInfo

IAMDroppedFrames Interface

Retrieves an array of frame numbers that the pin dropped.

HRESULT GetDroppedInfo(
  long *lSize***,**
  long * *plArray***,**
  long * *plNumCopied* **);**

Parameters

lSize
[in] Requested number of elements in the array.
plArray
[out] Pointer to the array.
plNumCopied
[out, retval] Pointer to the number of array elements filled in. This number can differ from *lSize* because the filter determines an arbitrary number of elements to save and it might not save this information for as many frames as you requested.

Return Values

Returns an HRESULT value that depends on the implementation of the interface.

Remarks

The filter will fill the array with the frame numbers of up to the first *lSize* number of frames dropped, and it will set *plNumCopied* accordingly.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

IAMDroppedFrames::GetNumDropped

IAMDroppedFrames Interface

Retrieves the total number of frames that the pin dropped since it last started streaming.

HRESULT GetNumDropped(
  long * *plDropped* **);**

Parameters

plDropped
  [out] Pointer to the total number of dropped frames.

Return Values

Returns an HRESULT value that depends on the implementation of the interface.

Previous Home Topic Contents Index Next

IAMDroppedFrames::GetNumNotDropped

IAMDroppedFrames Interface

Retrieves the total number of frames that the pin delivered downstream (did not drop).

HRESULT GetNumNotDropped(
  long * *plNotDropped* **);**

Parameters

plNotDropped
[out] Pointer to the total number of frames that weren't dropped.

Return Values

Returns an HRESULT value that depends on the implementation of the interface.

IAMExtDevice Interface

The **IAMExtDevice** interface is the base interface for controlling external devices. Developers can implement this interface to control numerous types of devices; however, the current DirectShow implementation is specific to VCRs. The **IAMExtDevice** interface controls general settings of external hardware and is intended to be used in combination with the IAMExtTransport interface, which controls a VCR's more specific settings. You can also implement the IAMTimecodeReader, IAMTimecodeGenerator, and IAMTimecodeDisplay interfaces if your filter manages SMPTE (Society of Motion Picture and Television Engineers) timecode, and the external device has the appropriate features.

For a description of a sample filter which controls a VCR through DirectShow, see Vcrctrl Sample (VCR Control Filter).

When to Implement

Implement this interface when you want to build a filter or application that controls an external device, such as a VCR. Because this interface controls general information about a device, implement the IAMExtTransport interface in addition to control the external device's more specific properties.

An application can directly instantiate and control external devices, such as VCRs, but it is strongly recommended that you always instantiate these devices within the context of a filter graph, even if they are the only filters within the graph.

When to Use

Use this interface when you want to add external device control to your application.

Applications should use the filter graph to enumerate the filters and then get the IAMExtDevice interface directly from the appropriate filter.

Hardware Requirements

To control an external VCR, certain hardware requirements are recommended. VCRs with an

RS-422 serial interface require a special serial port card or an external RS-232-to-RS-422 adapter. In addition, for best performance, your computer should have a serial port card built with a 16,550 high-performance UART (Universal Asynchronous Receiver/Transmitter) to sustain higher baud rates, such as 38.4 baud.

Methods in Vtable Order

IUnknown Methods	**Description**
QueryInterface	Retrieves pointers to supported interfaces.
AddRef	Increments the reference count.
Release	Decrements the reference count.

IAMExtDevice Methods	**Description**
GetCapability	Retrieves the capabilities of the external device.
get_ExternalDeviceID	Retrieves the model number of the external device.
get_ExternalDeviceVersion	Retrieves the version number of the external device's operating software.
put_DevicePort	Specifies the communication port to which the external device is connected.
get_DevicePower	Retrieves whether the external device's power mode is on, off, or standby.
put_DevicePower	Sets the external device's power mode to on, off, or standby.
Calibrate	Calibrates the external device's transport mechanism.
get_DevicePort	Retrieves the communication port to which the external device is connected.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

IAMExtDevice::Calibrate

IAMExtDevice Interface

Calibrates an external device's transport mechanism.

HRESULT Calibrate(
  HEVENT *hEvent*,
  long *Mode*,
  long **pStatus* **);**

Parameters

hEvent
    [in] Event used to signal completion of this process.

Mode
[in] Value that activates or deactivates the calibration process. Specify one of the following:

Value	**Meaning**
ED_ACTIVE	Activates calibration process.
ED_INACTIVE	Deactivates calibration process.
NULL	Used to determine if current status is active or inactive.

pStatus
[out] Value indicating whether an event is active (OATRUE) or inactive (OAFALSE).

Return Values

Returns an HRESULT value that depends on the implementation of the interface.

Remarks

Use this method on certain external devices that require calibration; for example, when rewinding a tape and resetting the counter, or computing the frame offset for a timecode reader or generator.

Filters for various external devices can implement this method differently, depending on the calibration that the device needs. This method assumes the IMediaEventSink interface has already established an event sink, or that another event signaling method has been established.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

IAMExtDevice::GetCapability

IAMExtDevice Interface

Retrieves the capabilities of the external device.

HRESULT GetCapability(
  long *Capability*,
  long **pValue*,
  double **pdblValue* **);**

Parameters

Capability
[in] Value that specifies which capability you want to check. This parameter must be one of the following values.

Value	Meaning
ED_DEVCAP_CAN_RECORD	Checks whether transport can record.
ED_DEVCAP_CAN_RECORD_STROBE	Checks whether transport can single-frame record.
ED_DEVCAP_CAN_SAVE	Checks whether transport can save data.
ED_DEVCAP_DEVICE_TYPE	Checks the external device type.
ED_DEVCAP_HAS_AUDIO	Checks whether transport has audio.
ED_DEVCAP_HAS_VIDEO	Checks whether the device has video.
ED_DEVCAP_USES_FILES	Checks whether transport has a built-in file system.

pValue

[out] Value indicating the capabilities of the property specified in the *Capability* parameter. Returns OATRUE if the property is supported or OAFALSE if the property is not supported for all properties except ED_DEVCAP_DEVICE_TYPE. In this case, returns one of the following:

Value	Meaning
ED_DEVTYPE_ATR	Audio Tape Recorder
ED_DEVTYPE_CG	Character Generator
ED_DEVTYPE_DDR	Digital Disc Recorder
ED_DEVTYPE_DVE	Digital video effects unit
ED_DEVTYPE_GPI	General purpose interface trigger
ED_DEVTYPE_KEYER	Video keyer
ED_DEVTYPE_LASERDISK	Laser disc
ED_DEVTYPE_MIXER_AUDIO	Audio mixer
ED_DEVTYPE_MIXER_VIDEO	Video mixer
ED_DEVTYPE_ROUTER	Video router
ED_DEVTYPE_TBC	Timebase corrector
ED_DEVTYPE_TCG	Timecode generator/reader
ED_DEVTYPE_VCR	VCR
ED_DEVTYPE_WIPEGEN	Video wipe generator
ED_DEVTYPE_JOYSTICK	Joystick
ED_DEVTYPE_KEYBOARD	Keyboard

pdblValue

[out] Value indicating the capabilities of the specified property (if it is a double value). Pass NULL if not in use.

Return Values

Returns an HRESULT value that depends on the implementation of the interface.

Remarks

All return values are in *pValue* unless you have large or floating point values to return, in which case they are returned in the *pdblValue* parameter.

IAMExtDevice::get_DevicePort

IAMExtDevice Interface

Retrieves the communication port to which the external device is connected.

HRESULT get_DevicePort(
  long **pDevicePort* **);**

Parameters

pDevicePort
[in] Port to which the device is connected. Retrieves one of the following:

Value	Meaning
DEV_PORT_1394	IEEE 1394 Bus
DEV_PORT_ARTI	ARTI driver
DEV_PORT_COM1	COM1
DEV_PORT_COM2	COM2
DEV_PORT_COM3	COM3
DEV_PORT_COM4	COM4
DEV_PORT_DIAQ	Diaquest driver
DEV_PORT_SIM	Simulation port
DEV_PORT_USB	Universal Serial Bus

Return Values

Returns an HRESULT value that depends on the implementation of the interface.

See Also

IAMExtDevice::put_DevicePort

IAMExtDevice::get_DevicePower

IAMExtDevice Interface

Retrieves the external device's power mode: on, off, or standby.

HRESULT get_DevicePower(
long **pPowerMode* **);**

Parameters

pPowerMode
[out] External device's power mode; can be one of the following values.

Value	Meaning
ED_POWER_OFF	Off
ED_POWER_ON	On
ED_POWER_STANDBY	Standby

Return Values

Returns an HRESULT value that depends on the implementation of the interface.

See Also

IAMExtDevice::put_DevicePower

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

IAMExtDevice::get_ExternalDeviceID

IAMExtDevice Interface

Retrieves the model number of the external device.

HRESULT get_ExternalDeviceID(
LPOLESTR **ppszData* **);**

Parameters

ppszData
[out] Returns the manufacturer-specific identification number or text as a string.

Return Values

Returns an HRESULT value that depends on the implementation of the interface.

Previous Home Topic Contents Index Next

IAMExtDevice::get_ExternalDeviceVersion

IAMExtDevice Interface

Retrieves the version number of the external device's operating software.

HRESULT get_ExternalDeviceVersion(
  LPOLESTR **ppszData* **);**

Parameters

ppszData
  [out] Returns the manufacturer-specific operating software version number from the external device.

Return Values

Returns an HRESULT value that depends on the implementation of the interface.

Previous Home Topic Contents Index Next

IAMExtDevice::put_DevicePort

IAMExtDevice Interface

Specifies the communication port to which the external device is connected.

HRESULT put_DevicePort(
  long *DevicePort*
  );

Parameters

DevicePort

[in] Port to which the device will connect. Specify one of the following:

Value	Meaning
DEV_PORT_1394	IEEE 1394 Bus
DEV_PORT_ARTI	ARTI driver
DEV_PORT_COM1	COM1
DEV_PORT_COM2	COM2
DEV_PORT_COM3	COM3
DEV_PORT_COM4	COM4
DEV_PORT_DIAQ	Diaquest driver
DEV_PORT_MIN	DEV_PORT_SIM
DEV_PORT_SIM	Simulation port (used for "no hardware" simulation)
DEV_PORT_USB	Universal serial bus

Return Values

Returns an HRESULT value that depends on the implementation of the interface.

See Also

IAMExtDevice::get_DevicePort

IAMExtDevice::put_DevicePower

IAMExtDevice Interface

Sets the external device's power mode to either on, off, or standby.

HRESULT put_DevicePower(
  long *PowerMode* **);**

Parameters

PowerMode

[in] Value indicating which power mode the device will have. Set to one of the following:

Value	**Meaning**
ED_POWER_OFF	Off
ED_POWER_ON	On
ED_POWER_STANDBY	Standby

Return Values

Returns an HRESULT value that depends on the implementation of the interface.

See Also

IAMExtDevice::get_DevicePower

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

IAMExtTransport Interface

The **IAMExtTransport** interface provides methods which control specific behaviors of an external VCR. These methods generally set and get the transport properties which relate to how the VCR and the computer exchange data. Since this interface controls specific behaviors of transport, it must be implemented in combination with the IAMExtDevice interface, which controls an external device's general behaviors. If you want to control an external device other than a VCR, two options are available. Either use the methods you need and return E_NOTIMPL for the rest, or design a new interface and aggregate it with **IAMExtDevice**.

This interface also provides methods that enable developers to define *edit events* which assist in the content authoring process. Edit events are made up of individual edit properties that are grouped together into *edit property sets*. These edit property sets can define an actual recording sequence on the transport or a simple positional command. They can, for example, specify certain modes of editing, record inpoints and outpoints, or memorize positions called bookmarks. The SetEditPropertySet method creates or registers a group of edit properties, called an edit property set, while the SetEditProperty enables the application to define parameters and values of individual edit properties. Since these are relatively sophisticated situations, their implementation is left to the advanced developer.

For a description of a sample filter which controls a VCR through DirectShow, see Vcrctrl Sample (VCR Control Filter).

When to Implement

Implement this interface if you want to build a filter or application that controls an external device, such as a VCR. Because this interface controls specific information about a device, you should implement it with the IAMExtDevice interface.

An application can directly instantiate and control external device control filters, such as those for VCRs, but it is strongly recommended that you always instantiated them within the context of a filter graph, even if they are the only filter within the graph.

Implementations can vary depending on the type of external device being controlled. With certain devices, methods can return E_NOTIMPL if they are not applicable.

When to Use

Use this interface if you want a filter to control video and audio tape machines that are external to the computer. Typical uses for this interface include the applications that implement "batch capture" and "print to tape" of audio and video.

Applications should use the filter graph to enumerate the filters and then get the IAMExtTransport interface directly from the appropriate filter.

Hardware Requirements

To control an external VCR, certain hardware requirements are recommended. VCRs with an RS-422 serial interface require a special serial port card or an external RS-232-to-RS-422 adapter. In addition, for best performance, your computer should have a serial port card built with a 16,550 high-performance UART to sustain higher baud rates, such as 38.4 baud.

Methods in Vtable Order

IUnknown methods	Description
QueryInterface	Retrieves pointers to supported interfaces.
AddRef	Increments the reference count.
Release	Decrements the reference count.

IAMExtTransport methods	Description
GetCapability	Retrieves the general capabilities of an external transport.
put_MediaState	Sets the current state of the media.
get_MediaState	Retrieves the current state of the media.
put_LocalControl	Sets the state of the external device to local or remote control.
get_LocalControl	Retrieves the state of the external device.
GetStatus	Determines the status of the external transport.
GetTransportBasicParameters	Retrieves the external transport's basic parameter settings.
SetTransportBasicParameters	Sets the external transport's basic parameters.
GetTransportVideoParameters	Retrieves the external transport's video parameter settings.
SetTransportVideoParameters	Sets the video parameters for the external transport.
GetTransportAudioParameters	Retrieves the external transport's audio parameter settings.
SetTransportAudioParameters	Sets audio parameter setting for the external transport.
put_Mode	Sets the movement of the transport to a new mode (play, stop, record, edit, and so on).
get_Mode	Retrieves the mode of the transport (play, stop, record, edit, and so on).
put_Rate	Sets the playback rate for variable-speed external devices.
get_Rate	Retrieves the playback rate set in put_Rate for variable speed external devices.

GetChase	Retrieves the status of chase mode.
SetChase	Enables or disables chase mode.
GetBump	Retrieves status of bump mode.
SetBump	Temporarily changes the speed of playback for synchronization of multiple external devices.
get_AntiClogControl	Determines if the anti-headclog control is enabled or disabled.
put_AntiClogControl	Enables or disables the transport's anti-headclog control.
GetEditPropertySet	Retrieves the current state of an edit property set.
SetEditPropertySet	Registers an edit property set that describes a group of edit properties.
GetEditProperty	Retrieves individual parameters and values associated with a particular edit property set.
SetEditProperty	Defines individual parameters and values associated with a particular edit property set.
get_EditStart	Determines if the external transport's edit control is active.
put_EditStart	Activates edit control on a capable transport.

Previous Home Topic Contents Index Next

IAMExtTransport::get_AntiClogControl

IAMExtTransport Interface

Determines if the anti-headclog control is enabled or disabled.

HRESULT get_AntiClogControl(
  long **pEnabled* **);**

Parameters

pEnabled
    [out] OATRUE indicates anti-headclog is enabled; OAFALSE indicates disabled.

Return Values

Returns an HRESULT value that depends on the implementation of the interface.

See Also

IAMExtTransport::put_AntiClogControl

IAMExtTransport::GetBump

IAMExtTransport Interface

Retrieves the status of bump mode.

HRESULT GetBump(
  long **pSpeed***,**
  long **pDuration* **);**

Parameters

pSpeed
[out] Temporary speed (a multiple of normal speed).

pDuration
[out] Pointer to the duration of a bump in current time format.

Return Values

Returns an HRESULT value that depends on the implementation of the interface.

Remarks

This method will cause a temporary speed variation of transport used during the physical synchronization process. It will stay in effect until *pDuration* time has expired.

See "IAMExtTransport Basic Parms" in DXMedia\Include\Edevdefs.h for supported time formats.

See Also

IAMExtTransport::SetBump

IAMExtTransport::GetCapability

IAMExtTransport Interface

Retrieves the general capabilities of an external transport.

HRESULT GetCapability(
  long *Capability*,
  long **pValue*,
  double **pdblValue* **);**

Parameters

Capability
  [in] Capability to query for. Specify one of the following:

Value	**Meaning**
ED_TRANSCAP_CAN_BUMP_PLAY	Checks whether transport can vary speed for synchronizing.
ED_TRANSCAP_CAN_DELAY_AUDIO_IN	Checks whether transport does delayed-in audio edits.
ED_TRANSCAP_CAN_DELAY_AUDIO_OUT	Checks whether transport does delayed-out audio edits.
ED_TRANSCAP_CAN_DELAY_VIDEO_IN	Checks whether transport does delayed-in video edits.
ED_TRANSCAP_CAN_DELAY_VIDEO_OUT	Checks whether transport does delayed-out video edits.
ED_TRANSCAP_CAN_EJECT	Checks whether transport can eject its media.
ED_TRANSCAP_CAN_PLAY_BACKWARDS	Checks whether transport can play media in reverse (negative rate).
ED_TRANSCAP_CAN_SET_EE	Checks whether transport can show its input on its output.
ED_TRANSCAP_CAN_SET_PB	Checks whether transport can show media playback on its output.
ED_TRANSCAP_FWD_VARIABLE_MAX	Maximum forward speed (multiple of play speed) in *pdblValue*.
ED_TRANSCAP_LTC_TRACK	Track number of linear timecode (LTC) in *pValue*.
ED_TRANSCAP_NUM_AUDIO_TRACKS	Number of audio tracks in *pValue*.
ED_TRANSCAP_REV_VARIABLE_MAX	Maximum reverse speed (multiple of play speed) in *pdblValue*.

pValue
  [out] Indicates whether the capability specified in *Capability* is supported or not. Returns either OATRUE if it is supported or OAFALSE if not.

pdblValue
  [out] Indicates the capabilities of the property specified in *Capability*.

Return Values

Returns an HRESULT value that depends on the implementation of the interface.

Remarks

All OATRUE and OAFALSE values are returned in *pValue*; numerical values are returned in *pValue* or *pdblValue*. Use the *pdblValue* parameter to return double values if the *pValue* parameter is insufficient. Return NULL if one of the parameters is not needed.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

IAMExtTransport::GetChase

IAMExtTransport Interface

Retrieves the status of chase mode.

HRESULT GetChase(
 long **pEnabled***,**
 long **pOffset***,**
 HEVENT **phEvent* **);**

Parameters

pEnabled
 [out] OATRUE specifies chase enabled; OAFALSE specifies chase disabled.
pOffset
 [out] Offset from the present time to which the transport will maintain while playing.
phEvent
 [out] Pointer to the completion notification that will signal chase offset is established.

Return Values

Returns an HRESULT value that depends on the implementation of the interface.

Remarks

The time for *pOffset* is given in the current time format (see "IAMExtTransport Basic Parms" in DXMedia\Include\Edevdefs.h for supported time formats).

See Also

IAMExtTransport::SetChase

Previous Home Topic Contents Index Next

IAMExtTransport::GetEditProperty

IAMExtTransport Interface

Retrieves the parameters and values associated with a particular edit event.

HRESULT GetEditProperty(
  long *EditID*,
  long *Param*,
  long **pValue* **);**

Parameters

EditID
  [in] Identification number of the edit property set.
Param
  [in] Edit event parameter to determine the value of.
pValue
  [out] Returns the value of the parameter specified in *Param*: OATRUE, OAFALSE, or a specific value.

Return Values

Returns an HRESULT value that depends on the implementation of the interface.

See Also

IAMExtTransport::SetEditProperty

Previous Home Topic Contents Index Next

IAMExtTransport::GetEditPropertySet

IAMExtTransport Interface

Retrieves individual parameters and values associated with a particular edit property set.

HRESULT GetEditPropertySet(
long *EditID*,
long **pState* **);**

Parameters

EditID
[in] Identification number of the edit property set.

pState
[out] State of the edit property set. Retrieves one of the following:

Value	Meaning
DC_SET_ACTIVE	Activates edit property set.
DC_SET_DELETE	Deletes edit property set.
DC_SET_INACTIVE	Deactivates edit property set.
DC_SET_REGISTER	Registers edit property set.

Return Values

Returns an HRESULT value that depends on the implementation of the interface.

See Also

IAMExtTransport::SetEditPropertySet

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

IAMExtTransport::get_EditStart

IAMExtTransport Interface

Determines if the external transport's edit control is active.

HRESULT get_EditStart(
long **pValue* **);**

Parameters

pValue

[out] Returns OATRUE if edit control is active; OAFALSE if inactive.

Return Values

Returns an HRESULT value that depends on the implementation of the interface.

See Also

IAMExtTransport::put_EditStart

Previous Home Topic Contents Index Next

IAMExtTransport::get_LocalControl

IAMExtTransport Interface

Retrieves the state of the external device.

HRESULT get_LocalControl(
long **pState* **);**

Parameters

pState
[out] Returns either OATRUE for local control or OAFALSE for remote.

Return Values

Returns an HRESULT value that depends on the implementation of the interface.

Remarks

To control an external device, it must be in remote mode.

See Also

IAMExtTransport::put_LocalControl

Previous Home Topic Contents Index Next

IAMExtTransport::get_MediaState

IAMExtTransport Interface

Retrieves the current state of the media set in put_MediaState.

HRESULT get_MediaState(
  long **pState* **);**

Parameters

pState
[out] Returns the current state or the media. Values will be device specific but can include the following:

Value	Meaning
ED_MEDIA_SPIN_DOWN	Stop spinning (for disk media); unthread the tape (for tape media).
ED_MEDIA_SPIN_UP	Start spinning (for disk media); thread the tape (for tape media).
ED_MEDIA_UNLOAD	Eject the media from the drive (if device supports it).

Return Values

Returns an HRESULT value that depends on the implementation of the interface.

Previous Home Topic Contents Index Next

IAMExtTransport::get_Mode

IAMExtTransport Interface

Retrieves the mode of the transport (play, stop, record, edit, and so on).

HRESULT get_Mode(
  long **pMode* **);**

Parameters

pMode
[out] Current transport mode (see IAMExtTransport::put_Mode for possible modes).

Return Values

Returns an HRESULT value that depends on the implementation of the interface.

IAMExtTransport::get_Rate

IAMExtTransport Interface

Retrieves the playback rate for variable-speed external devices.

HRESULT get_Rate(
  double **pdblRate* **);**

Parameters

pdblRate
  [out] Pointer to the playback rate set in IAMExtTransport::put_Rate.

Return Values

Returns an HRESULT value that depends on the implementation of the interface.

IAMExtTransport::GetStatus

IAMExtTransport Interface

Determines the external transport's status.

HRESULT GetStatus(
  long *StatusItem*,
  long **pValue* **);**

Parameters

StatusItem

[in] Item you want to determine the status of; can include one of the following:

Value	Meaning
ED_MODE_EDIT_CUE	Checks if device is cueing for an active edit event.
ED_MODE_FF	Checks if device is fast forwarding.
ED_MODE_FREEZE	Checks if device is paused.
ED_MODE_LINK_OFF	Checks if transport control isn't linked to filter graph's run, stop, and pause controls.
ED_MODE_LINK_ON	Checks if transport control is linked to filter graph's run, stop, pause controls.
ED_MODE_PLAY	Checks if device is playing.
ED_MODE_RECORD	Checks if device is recording.
ED_MODE_RECORD_STROBE	Checks if device is recording single-frame.
ED_MODE_REW	Checks if device is rewinding.
ED_MODE_SHUTTLE	Checks if device is shuttling (high-speed movement with visible picture).
ED_MODE_STEP	Checks if device is single-stepping.
ED_MODE_STOP	Checks if device is stopped.

pValue

[out] Returns OATRUE if *StatusItem* is active or OAFALSE if not.

Return Values

Returns an HRESULT value that depends on the implementation of the interface.

Remarks

When implementing this interface, be aware that transport *StatusItem* parameters are more extensive than most Microsoft® DirectShow™ interfaces and code should reflect this variety and feel free to check the transport status of appropriate parameters.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

IAMExtTransport::GetTransportAudioParameters

IAMExtTransport Interface

Retrieves audio parameter setting for external transport.

HRESULT GetTransportAudioParameters(

long *Param*,
long **pValue* **);**

Parameters

Param
[in] Audio parameter whose value you want to get. Specify one of the following:

Value	Meaning
ED_TRANSAUDIO_ENABLE_OUTPUT	Audio output channel(s)
ED_TRANSAUDIO_ENABLE_RECORD	Audio recording channel(s)
ED_TRANSAUDIO_ENABLE_SELSYNC	Audio selsync recording channel(s)
ED_TRANSAUDIO_SET_MONITOR	Monitor output audio channel(s)
ED_TRANSAUDIO_SET_SOURCE	Audio source channel(s)

pValue
[out] Channel or channels set in the IAMExtTransport::SetTransportAudioParameters method.

Return Values

Returns an HRESULT value that depends on the implementation of the interface.

IAMExtTransport::GetTransportBasicParameters

IAMExtTransport Interface

Retrieves the transport's basic parameter settings.

HRESULT GetTransportBasicParameters(
long *Param*,
long **pValue*,
LPOLESTR **ppszData* **);**

Parameters

Param
[in] Parameter you want to receive the setting for (see Edevdefs.h for possible parameters under "IAMExtTransport Basic Parms").
pValue
[out] Setting of the parameter if it is numeric.
ppszData
[out] Setting of the parameter if it is a string .

Return Values

Returns an HRESULT value that depends on the implementation of the interface.

See Also

IAMExtTransport::SetTransportBasicParameters

IAMExtTransport::GetTransportVideoParameters

IAMExtTransport Interface

Retrieves video parameter settings for external transport.

HRESULT GetTransportVideoParameters(
  long *Param*,
  long **pValue* **);**

Parameters

Param
  [in] Video parameter you want to receive the settings for. Can be either ED_TRANSVIDEO_SET_OUTPUT (video transport output parameters) or ED_TRANSVIDEO_SET_SOURCE (video transport source).

pValue
  [out] Set the ED_TRANSVIDEO_SET_SOURCE flag to retrieve the active video input, or set the ED_TRANSVIDEO_SET_OUTPUT flag to retrieve one of the following values:

Value	Meaning
ED_E2E	Input video is visible on device's output regardless of transport mode.
ED_OFF	Video output is disabled.
ED_PLAYBACK	Video playing from media is displayed on the screen.

Return Values

Returns an HRESULT value that depends on the implementation of the interface.

See Also

IAMExtTransport::SetTransportVideoParameters

IAMExtTransport::put_AntiClogControl

IAMExtTransport Interface

Enables or disables transport anti-headclog control.

HRESULT put_AntiClogControl(
  long *Enable* **);**

Parameters

Enable
    [in] Value indicating whether to enable anti-headclog control; set OATRUE to enable, OAFALSE to disable.

Return Values

Returns an HRESULT value that depends on the implementation of the interface.

Remarks

Use this method to unclog video heads on VCRs that have an automatic head-cleaning feature.

See Also

get_AntiClogControl

IAMExtTransport::put_EditStart

IAMExtTransport Interface

Activates the edit control on a capable transport.

HRESULT put_EditStart(
  long *Value* **);**

Parameters

Value
  [in] OATRUE activates the edit control; OAFALSE deactivates it.

Return Values

Returns an HRESULT value that depends on the implementation of the interface.

Remarks

Use this method to manually enable edit control. Edit control is defined as the precise enabling of individual, or a set of, record tracks on a VCR; for example, a video-only insert edit, where only the video record head is enabled and a new video signal is recorded — the audio signal is left as is. Use this method to control "on the fly" editing on machines that have this feature.

See Also

IAMExtTransport::get_EditStart

IAMExtTransport::put_LocalControl

IAMExtTransport Interface

Sets the state of the external device to local or remote control.

HRESULT put_LocalControl(
  long *State* **);**

Parameters

State
  [in] Current state; pass OATRUE for local, OAFALSE for remote.

Return Values

Returns an HRESULT value that depends on the implementation of the interface.

See Also

IAMExtTransport::get_LocalControl

Previous Home Topic Contents Index Next

IAMExtTransport::put_MediaState

IAMExtTransport Interface

Sets the current state of the media.

HRESULT put_MediaState(
  long *State* **);**

Parameters

State
[in] Value specifying the state. Use one of the following:

Value	**Meaning**
ED_MEDIA_SPIN_DOWN	Stop spinning (for disc media); unthread the tape (for tape media).
ED_MEDIA_SPIN_UP	Start spinning (for disc media); thread the tape (for tape media).
ED_MEDIA_UNLOAD	Eject the media from the drive (if device supports it).

Return Values

Returns an HRESULT value that depends on the implementation of the interface.

Remarks

Use the preceding parameters for disk and tape media. For other devices, you might need to specify new parameters.

See Also

IAMExtTransport::get_MediaState

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

IAMExtTransport::put_Mode

IAMExtTransport Interface

Sets the movement of the transport to a new mode (play, stop, record, edit, and so on).

HRESULT put_Mode(
 long *Mode* **);**

Parameters

Mode
[in] Transport mode. Specify one of the following:

Value	Meaning
ED_MODE_PLAY	Play
ED_MODE_STOP	Stop
ED_MODE_FREEZE	Freeze (pause)
ED_MODE_THAW	Resume
ED_MODE_FF	Fast forward
ED_MODE_REW	Rewind
ED_MODE_RECORD	Record
ED_MODE_RECORD_STROBE	Record single frame
ED_MODE_STEP	Single step
ED_MODE_SHUTTLE	Shuttle (high-speed movement with visible picture)
ED_MODE_EDIT_CUE	Cue for an edit event
ED_MODE_LINK_ON	Link this method to the graph's IMediaControl::Run, Stop, and Pause methods
ED_MODE_LINK_OFF	Disengage this method from the graph's IMediaControl::Run, Stop, Pause methods.

Return Values

Returns an HRESULT value that depends on the implementation of the interface.

See Also

IAMExtTransport::get_Mode

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

IAMExtTransport::put_Rate

IAMExtTransport Interface

Sets the playback rate for variable-speed external devices.

HRESULT put_Rate(
  double *dblRate* **);**

Parameters

dblRate
 [in] Multiple of play speed where .5=half, 1=normal, 2=double, 3=triple and so forth.

Return Values

Returns an HRESULT value that depends on the implementation of the interface.

Remarks

This method enables an application to speed up or slow down playback relative to the normal default playback speed. A rate of 1.0 indicates normal playback speed. Specifying 2.0 causes playback at twice the normal rate.

You can also link this method to the IMediaPosition::put_Rate method as an alternate means of setting rates of playback relative to normal speed.

See Also

IAMExtTransport::get_Rate

Previous Home Topic Contents Index Next

IAMExtTransport::SetBump

IAMExtTransport Interface

Temporarily changes the speed of playback for synchronization of multiple external devices.

HRESULT SetBump(
  long *Speed*,

long *Duration* **);**

Parameters

Speed
[in] Temporary speed (a multiple of normal speed).
Duration
[in] Duration of a bump in current time format.

Return Values

Returns an HRESULT value that depends on the implementation of the interface.

Remarks

This method will stay in effect until *Duration* time has expired.

See "IAMExtTransport Basic Parms" in DXMedia\Include\Edevdefs.h for supported time formats.

See Also

IAMExtTransport::GetBump

IAMExtTransport::SetChase

IAMExtTransport Interface

Enables or disables chase mode.

HRESULT SetChase(
long *Enable*,
long *Offset*,
HEVENT *hEvent* **);**

Parameters

Enable
[in] Enables or disables chase. Specify OATRUE to enable chase; OAFALSE to disable.
Offset
[in] Offset from the time reference that the transport will maintain. Specify in current

time format.

hEvent
: [in] Event to signal offset established.

Return Values

Returns an HRESULT value that depends on the implementation of the interface.

Remarks

Use the **SetChase** method when you want an external transport to continuously follow a timecode signal with a fixed offset. For example, if your computer is generating timecode, a VCR capable of chasing can be told by the computer to put itself in play mode and keep its media a fixed offset from the reference timecode. You determine the offset by comparing the timecode on the playback media to the reference (generated) timecode.

This method will stay in effect until canceled or complete and requires the filter to verify (by periodically reading the transport's timecode) that the transport is indeed maintaining the fixed offset.

Time for *Offset* is specified in current time format (see "IAMExtTransport Basic Parms" in DXMedia\Include\Edevdefs.h for supported time formats).

See Also

IAMExtTransport::GetChase

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

IAMExtTransport::SetEditProperty

IAMExtTransport Interface

Defines individual parameters and values associated with a particular edit property set.

HRESULT SetEditProperty(
  long *EditID*,
  long *Param*,
  long *Value* **);**

Parameters

EditID
: [in] Identification number of the edit property set.

Param
[in] Edit event parameter to define.
Value
[in] Value of the parameter specified in *Param*. Use OATRUE, OAFALSE, or a specific value.

Return Values

Returns an HRESULT value that depends on the implementation of the interface.

Remarks

Edit events can either refer to a group of predefined properties that define an actual recording sequence, or they can refer to simple positional commands. They can, for example, specify certain modes of editing, record inpoints and outpoints, or memorize positions called bookmarks. The SetEditPropertySet method defines and registers a group of edit events, called an edit property set, while the **SetEditProperty** method enables the user to define parameters and values of individual edit events.

To define a set of edit properties, first register an edit property set and get an *EditID* with the SetEditPropertySet method. Then use the **SetEditProperty** method to define specific parameters and values of individual edit properties. Finally, use the **SetEditPropertySet** method to activate the edit property set.

For a complete listing of possible parameters and values for edit property sets see Edevdefs.h in the DirectShow SDK's DXMedia\Include folder.

See Also

IAMExtTransport::GetEditProperty

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

IAMExtTransport::SetEditPropertySet

IAMExtTransport Interface

Registers an edit property set that describes a group of edit properties.

HRESULT SetEditPropertySet(
  long **pEditID*,
  long *State* **);**

Parameters

pEditID
[in, out] Identification number of the edit property set.
State
[in] State of the edit property set. Specify one of the following:

Value	Meaning
DC_SET_ACTIVE	Activates edit property set.
DC_SET_DELETE	Deletes edit property set.
DC_SET_INACTIVE	Inactivates edit property set.
DC_SET_REGISTER	Registers edit property set.

Return Values

Returns an HRESULT value that depends on the implementation of the interface.

Remarks

Edit events refer to a group of predefined properties that define an actual recording sequence on the transport or a simple positional command. They can, for example, specify certain modes of editing, record inpoints and outpoints, or memorize positions called bookmarks. The **SetEditPropertySet** method defines and registers a group of edit properties, called an edit property set, while the SetEditProperty enables the user to define parameters and values of individual edit event properties.

To define a set of edit properties, first register an edit property set and get an *EditID* with the **SetEditPropertySet** method. Then use the SetEditProperty method to define specific parameters and values of individual edit properties. Finally, use the **SetEditPropertySet** method to activate the edit property set.

For a complete listing of possible parameters and values for edit property sets see Edevdefs.h in the DirectShow SDK's DXMedia\Include folder.

See Also

IAMExtTransport::GetEditPropertySet

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

IAMExtTransport::SetTransportAudioParameters

IAMExtTransport Interface

Sets audio parameter setting for external transport.

HRESULT SetTransportAudioParameters(
  long *Param*,
  long *Value* **);**

Parameters

Param
[in] Audio parameter you want to set. Specify one of the following.

Value	Meaning
ED_TRANSAUDIO_ENABLE_OUTPUT	Enable audio channel(s) for output.
ED_TRANSAUDIO_ENABLE_RECORD	Enable audio channel(s) for recording.
ED_TRANSAUDIO_ENABLE_SELSYNC	Enable audio channel(s) for selsync recording.
ED_TRANSAUDIO_SET_MONITOR	Set the monitor output source.
ED_TRANSAUDIO_SET_SOURCE	Set the active audio input.

Value
[in] Audio channel or channels to set the parameter on.

Return Values

Returns an HRESULT value that depends on the implementation of the interface.

Remarks

Specify an exact channel or channels in *Value* by selecting ED_AUDIO_1 through ED_AUDIO_24 (use an **or** switch to combine), or all channels by selecting ED_AUDIO_ALL.

See Also

IAMExtTransport::GetTransportAudioParameters

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

IAMExtTransport::SetTransportBasicParameters

IAMExtTransport Interface

Sets basic parameters of external transport.

HRESULT SetTransportBasicParameters(
  long *Param*,
  long *Value*,

  LPCOLESTR *pszData* **);**

Parameters

Param
: [in] Parameter you want to set (see Edevdefs.h for possible parameters under "IAMExtTransport Basic Parms").

Value
: [in] Setting of the parameter if it is numeric.

pszData
: [in] Setting of the parameter if it is a string.

Return Values

Returns an HRESULT value that depends on the implementation of the interface.

Remarks

Basic settings include time formats, record formats, preroll setting, servo setting, and others (see Edevdefs.h).

See Also

IAMExtTransport::GetTransportBasicParameters

IAMExtTransport::SetTransportVideoParameters

IAMExtTransport Interface

Sets video parameters for external transport.

HRESULT SetTransportVideoParameters(
  long *Param*,
  long *Value* **);**

Parameters

Param
: [in] Video parameter you want to set. Specify either ED_TRANSVIDEO_SET_OUTPUT (video transport output parameters) or ED_TRANSVIDEO_SET_SOURCE (video transport source).

Value

[in] Set the ED_TRANSVIDEO_SET_SOURCE flag to set the active video input, or set the ED_TRANSVIDEO_SET_OUTPUT flag to one of the following values.

Value	Meaning
ED_E2E	Input video is visible on device's output regardless of transport mode.
ED_OFF	Video output is disabled.
ED_PLAYBACK	Video playing from media is displayed on the screen.

Return Values

Returns an HRESULT value that depends on the implementation of the interface.

Remarks

For the ED_TRANSVIDEO_SET_SOURCE flag, an external device filter uses physical pins to describe its inputs. Calls to the filter's pin enumerator will return an index value. The value is passed to this method as its as its *Value* parameter.

See Also

IAMExtTransport::GetTransportVideoParameters

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

IAMFileCutListElement Interface

The **IAMFileCutListElement** interface provides support for a cutlist element for a file stream.

See About Cutlists and Using Cutlists for more information.

When to Implement

Implement this interface in your application when you implement your own IAMCutListElement interface. Usually, you don't need to implement either interface because DirectShow provides the CLSID_VideoFileClip and CLSID_AudioFileClip objects that implement it for you. However, you can implement this interface in your application when you need to change the default behavior of this interface.

When to Use

Use this interface in your filter when you specify a media clip stored in a file. Call QueryInterface on IAMCutListElement to determine if the element is file-based.

When compiling a cutlist application you must explicitly include the cutlist header file as follows:

```
#include <cutlist.h>
```

Methods in Vtable Order

IUnknown methods	**Description**
QueryInterface	Retrieves pointers to supported interfaces.
AddRef	Increments the reference count.
Release	Decrements the reference count.

IAMFileCutListElement methods	**Description**
GetFileName	Retrieves the file name of the cutlist element.
GetTrimInPosition	Retrieves the media time of the trimin point, based on the timeline of the cut's source file.
GetTrimOutPosition	Retrieves the media time of the trimout point, based on the timeline of the cut's source file.
GetOriginPosition	Retrieves the media time of the origin of the file or clip.
GetTrimLength	Retrieves the length of time between the trimin and trimout points.
GetElementSplitOffset	Retrieves the media time of the number of frames between the trimin point and the start of this element in output time.

Previous Home Topic Contents Index Next

Previous Home Topic Contents Index Next

IAMFileCutListElement::GetElementSplitOffset

IAMFileCutListElement Interface

Retrieves the media time of the number of frames between the trimin point and the start of this element in output time.

HRESULT GetElementSplitOffset(
  REFERENCE_TIME **pmtOffset*
  );

Parameters

pmtOffset
[out] Pointer that will receive the offset in the element's length.

Return Values

Returns an HRESULT value that depends on the implementation of the interface. **HRESULT** can include one of the following standard constants, or other values not listed.

Value	**Meaning**
E_FAIL	Failure.
E_INVALIDARG	Argument is invalid.
E_NOTIMPL	Method is not supported.
E_POINTER	Null pointer argument.
S_OK	Success.

Remarks

This method must retrieve a zero offset. Other offsets are not supported.

IAMFileCutListElement::GetFileName

IAMFileCutListElement Interface

Retrieves the file name of the cutlist element.

HRESULT GetFileName(
 LPWSTR **ppwstrFileName*
 );

Parameters

ppwstrFileName
 [out] Pointer that will receive the file name (must be freed when no longer needed).

Return Values

Returns an HRESULT value that depends on the implementation of the interface. **HRESULT** can include one of the following standard constants, or other values not listed.

Value	**Meaning**
E_FAIL	Failure.
E_INVALIDARG	Argument is invalid.
E_NOTIMPL	Method is not supported.
E_POINTER	Null pointer argument.
S_OK	Success.

IAMFileCutListElement::GetOriginPosition

IAMFileCutListElement Interface

Retrieves the media time of the origin of the file or clip.

HRESULT GetOriginPosition(
  REFERENCE_TIME **pmtOrigin*
  );

Parameters

mtOrigin
[out] Pointer that will receive the origin. The origin is in media time.

Return Values

Returns an HRESULT value that depends on the implementation of the interface. **HRESULT** can include one of the following standard constants, or other values not listed.

Value	**Meaning**
E_FAIL	Failure.
E_INVALIDARG	Argument is invalid.
E_NOTIMPL	Method is not supported.
E_POINTER	Null pointer argument.
S_OK	Success.

Remarks

This method must return a zero origin. Clips with nonzero start times are not supported.